НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ І
СПЕЦІАЛІЗОВАНИХ КОМП'ЮТЕРНИХ СИСТЕМ

«На правах рукопису»
УДК 004.04

«До захисту допущено»

Завідувач кафедри СПСКС

_____

В.П.Тарасенко_
(підпис) (ініціали, прізвище)

"___"_____2018р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 123 Комп'ютерна інженерія
Комп'ютерні системи та компоненти

на тему: МЕТОД ПРИСКОРЕННЯ СТАТИСТИЧНОГО МОДЕЛЮВАННЯ МОВ ПРОГРАМУВАННЯ НА ОСНОВІ ВИКОРИСТАННЯ ФРЕЙМВОРКУ TENSORFLOW

Виконав: студент ІІ курсу, групи КВ-71мп
(шифр групи)

Гнідий Дмитро Олександрович _____ _____
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник доц., к.т.н. Марченко О. І. _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2018 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – другий (магістерський)
Спеціальність 123 Комп'ютерна інженерія
Комп'ютерні системи та компоненти

ЗАТВЕРДЖУЮ
Завідувач кафедри СПСКС
_____
_В.П.Тарасенко__
(підпис) (ініціали, прізвище)
«___»_____2018р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Гнідому Дмитру Олександровичу

1. Тема дисертації: МЕТОД ПРИСКОРЕННЯ СТАТИСТИЧНОГО МОДЕЛЮВАННЯ МОВ ПРОГРАМУВАННЯ НА ОСНОВІ ВИКОРИСТАННЯ ФРЕЙМВОРКУ TENSORFLOW,
науковий керівник дисертації: Марченко Олександр Іванович доцент кафедри СПіСКС, к.т.н., доцент,
затверджені наказом по університету від «30» жовтня 2018 р. №4030-с
2. Термін подання студентом дисертації: 7 грудня 2018 р.
3. Об'єкт дослідження: процес прискорення статистичного моделювання мов програмування при використанні рекурентної нейронної мережі
4. Предмет дослідження: методи прискорення статистичного моделювання мов програмування для гетерогенних паралельних архітектур
5. Перелік завдань, які потрібно розробити:
   - опис предметної області досліджень та обґрунтування методу прискорення статистичного моделювання мов програмування з використанням фреймворку TENSORFLOW;
   - алгоритм методу прискорення статистичного моделювання мов програмування з використанням фреймворку TENSORFLOW;
   - модифікований алгоритм уваги для рекурентної нейромережі з використанням фреймворку TENSORFLOW.
6. Перелік ілюстративного матеріалу: презентація (кількість аркушів: 12)

7. Перелік публікацій:

- «Адаптований механізм уваги рекурентної нейромережі для використання у фреймворку TENSORFLOW», міжнародна мультидисциплінарна конференція «Наука і техніка сьогодення: пріоритетні напрямки розвитку України та Польщі». – 2018;
- «Метод прискорення статичного моделювання мов програмування на основі використання фреймворку TENSORFLOW», XI конференція молодих вчених ПМК-2018-2. – 2018.

8. Дата видачі завдання: 5 вересня 2017 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Вивчення літератури за тематикою проекту	10.10.2017	
2	Розроблення та узгодження завдання магістерської дисертації	21.10.2018	
3	Аналіз існуючих підходів статистичного моделювання мов програмування	10.01.2018	
4	Тестування існуючих підходів статистичного моделювання мов програмування	25.02.2018	
5	Підготовка матеріалів першого розділу магістерської дисертації	16.04.2018	
6	Підготовка матеріалів другого розділу магістерської дисертації	05.06.2018	
7	Реалізація запропонованого підходу	13.09.2018	
8	Підготовка матеріалів третього розділу магістерської дисертації	27.07.2018	
9	Підготовка матеріалів четвертого розділу магістерської дисертації	13.09.2018	
10	Підготовка графічної частини дипломного проекту	16.10.2018	
11	Оформлення документації дипломного проекту	01.11.2018	
12	Попередній розгляд магістерської дисертації на кафедрі	26.11.2018	

Студент _____ Гнідий Д. О.
(підпис)

Науковий керівник дисертації _____ Марченко О. І.
(підпис)

Актуальність теми. Останнім часом використання нейронних мереж у розробці статистичних моделей мов програмування стало дуже популярним. За допомогою нейронних мереж досягаються кращі результати, ніж з класичними методами як в окремих моделях мов, так і в тому випадку, коли моделі входять у більші моделі у складних завданнях, таких як розпізнавання мовлення, машинний переклад чи генерація програм з намірів користувача. Ключовою причиною якісного підвищення продуктивності може бути здатність методу узагальнювати.

Тому дослідження методу прискорення статистичного моделювання мов програмування з використанням фреймворку TensorFlow є актуальною задачею у наш час.

Об'єктом дослідження є процес прискорення статистичного моделювання мов програмування при використанні рекурентної нейронної мережі.

Предметом дослідження є методи прискорення статистичного моделювання мов програмування для гетерогенних паралельних архітектур.

Мета роботи: прискорення статистичного моделювання мов програмування на основі використання фреймворку TensorFlow та обчислень на гетерогенних паралельних архітектур на базі рекурентної нейронної мережі, а також розробка програмної реалізації запропонованого методу.

Наукова новизна:

1. Проаналізовано існуючі методи та системи статистичного моделювання мов програмування і показано, що ці методи та системи мають недоліки у їх використанні за різними показниками: низька швидкість роботи та обмеженість варіантів використання.
2. Запропоновано метод прискорення статистичного моделювання мов програмування, який відрізняється від існуючих використанням

фреймворку TensorFlow та виконанням обчислень на гетерогенних паралельних архітектур на базі рекурентної нейронної мережі, а саме на платах TPU, що дозволило збільшити швидкість роботи системи в 2,5 рази і зменшити витрати на користування системою на 20% у порівняння з використанням GPU.

Практична цінність отриманих в роботі результатів полягає в тому, що запропонований метод та його програмна реалізація дають змогу більш швидше та продуктивніше проводити навчання нейронної мережі та отримувати більш якісну мовну модель з більш складною зв'язністю між окремими словами та реченнями у вхідному тексті, що дозволяє більш точно розпізнавати мови програмування та виокремлювати зміст написаного та генерувати вихідну програму з намірів користувача.

Апробація роботи. Метод прискорення статичного моделювання мов програмування на основі використання фреймворку TensorFlow представлений та обговорювався на XI науковій конференції молодих вчених "Прикладна математика та комп'ютинг" ПМК-2018-2 (Київ, 15-17 листопада 2018 р.). Адаптований механізм уваги рекурентної нейромережі для використання у фреймворку TensorFlow представлений та обговорювався на міжнародній мультидисциплінарній конференції «Наука і техніка сьогодення: пріоритетні напрямки розвитку України та Польщі» (м. Воломін, Польща, 19-20 жовтня 2018 р.).

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів та висновків.

У вступі подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих результатів і практичну цінність роботи.

У першому розділі розглянуто існуючі методи статистичного моделювання мов програмування, їхні особливості, недоліки та переваги, розглянуто різні реалізації.

У другому розділі запропоновано метод прискорення статистичного моделювання мов програмування на основі використання фреймворку TensorFlow.

У третьому розділі наведено особливості реалізації розробленої системи.

У четвертому розділі представлено підходи до тестування системи в цілому та окремих модулів.

У висновках представлені результати проведеної роботи.

Робота представлена на 80 аркушах, містить посилання на список використаних літературних джерел.

Ключові слова: статистичне моделювання мов, рекурентна нейромережа, механізм уваги, TPU, TensorFlow.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

CNN – Convolutional Neural Network

DNN – Deep Neural Network

GRU – Gated Recurrent Unit

LLVM – Low Level Virtual Machine

LSTM – Long short-term memory

ML – Machine Learning

NLP – Neuro-linguistic Programming

NMT – Neural Machine Translation

RNN – Recurrent Neural Network

SE – Software Engineering

TPU – Tensor Processing Unit

WebGL – Web Graphics Library

ВСТУП

Останнім часом використання нейронних мереж у розробці статичних моделей мов стало дуже популярним. За допомогою нейронних мереж досягаються кращі результати, ніж з класичними методами як в окремих моделях мов, так і в тому випадку, коли моделі входять у більші моделі у складних завданнях, таких як розпізнавання мовлення та машинний переклад. Ключовою причиною якісного підвищення продуктивності може бути здатність методу узагальнювати.

Основними завданнями статичних моделей мов є: машинний переклад, попередня обробка, виокремлення змісту, детектування або сегментація, високорівнева обробка.

Методи статичного моделювання мов[1] вирішують таку задачу: розпізнавати природні мови та виокремлювати зміст написаного, робити передбачення наступного слова у реченні.

Формальні мови, такі як мови програмування, можуть бути повністю уточнені, описані певним набором правил, граматикою. Усі зарезервовані слова можуть бути визначені, а їх взаємне розташування строго задане граматикою мови. Цього не можна зробити з природною мовою. Природні мови не розроблені, вони виникають у процесі розвитку людства, і тому немає офіційної специфікації. Хоч існують формальні правила для частин мови та евристики, але часто застосовується природна мова, яка не відповідає узгодженим правилам. Крім того, мови змінюються з часом, словосполучення змінюється: це не постійна одиниця. Тим не менше, лінгвісти намагаються визначати формальні граматики та структури у природних мовах. Це можна зробити, але це дуже складно, і результати можуть бути крихкими. Альтернативним підходом до визначення моделі мови є вивчення її на прикладах[2].

Дана робота ставить перед собою такі завдання: розробити метод прискорення статистичного моделювання мов на основі використання фреймворку TensorFlow та обчислень на гетерогенній паралельній архітектурі на базі використання TPU, а також розробити програмну реалізацію цього методу.

1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Огляд алгоритмів статистичного моделювання мов

1.1.1 Unigram моделі

Модель Unigram, яка використовується для пошуку інформації, може розглядатися як комбінація декількох одно стадійних кінцевих автоматів з одним станом. Вона розщеплює ймовірності різних термінів у контексті, наприклад, від (1.1) до (1.2).

$$P(t_1 t_2 t_3) = P(t_1)P(t_2|t_1)P(t_3|t_1 t_2), \quad (1.1)$$

$$P(t_1 t_2 t_3) = P(t_1)P(t_2)P(t_3). \quad (1.2)$$

У цій моделі ймовірність кожного слова залежить тільки від власної ймовірності цього слова в документі, тому ми маємо лише кінцеві автомати з єдиним станом. Сам автомат має розподіл імовірностей по всьому словнику моделі, підсумовуючи його до 1. Ілюстрація уніграмної моделі документа представлена у табл. 1.1.

Всі основні обчислення, навчання та читання даних абстракцій CNTK Python API дуже легко розширюються як з Python, так і з C ++, що дозволяє користувачам впровадження нових операторів, учнів та читачів даних, які вільно створюються за допомогою вбудованих засобів бібліотеки. API вводить новий формат серіалізації на базі протоколів Buffers, який підтримує зворотню сумісність збережених моделей.

Таблиця 1.1. Ілюстрація уніграмної моделі документа

Одиниці мови	Вірогідність у документі
a	0.1
world	0.2
likes	0.05
we	0.05
share	0.3

Вірогідність, сформована для конкретного запиту, розраховується за формулою

$$P(query) = \prod_{term\ in\ query} P(term). \quad (1.3)$$

Для різних документів ми можемо побудувати власні Unigram моделей, з різними ударами ймовірностей слів у ній. І ми використовуємо ймовірності з різних документів, щоб генерувати різну вірогідність попадання запиту. Потім ми можемо класифікувати документи для запиту відповідно до згенерованих імовірностей[3]. Наступний приклад двох Unigram моделей двох документів (табл. 1.2).

Таблиця 1.2. Приклад двох Unigram моделей двох документів

Одиниці мови	Вірогідність у док. 1	Вірогідність у док. 2
a	0.1	0.3
world	0.2	0.1

likes	0.05	0.03
we	0.05	0.02
share	0.3	0.2

У контекстах пошуку інформації Unigram мовні моделі часто згладжуються, щоб уникнути випадків, коли P (одиниці мови) = 0. Загальний підхід полягає в тому, щоб створити модель максимальної правдоподібності для всієї колекції та лінійно інтерполювати модель колекції з моделлю максимальної ймовірності для кожного документа.

1.1.2 N-gram моделі

В області обробки природної мови N-gram моделі використовується в основному для передбачення на основі імовірнісних моделей. N-gram модель розраховує ймовірність останнього слова N-gram послідовності, якщо відомі всі попередні. При використанні цього підходу для моделювання мови передбачається, що поява кожного слова залежить тільки від попередніх слів.

Іншим застосуванням N-грам є виявлення плагіату. Якщо розділити текст на кілька невеликих фрагментів, представлених N-gram моделями, їх легко порівняти один з одним і таким чином отримати ступінь подібності документів. N-gram моделі успішно використовуються для категоризації тексту і мови. Крім того, їх можна використовувати для створення функцій, які дозволяють отримувати знання з текстових даних. Використовуючи N-gram моделі, можна ефективно знайти кандидатів, щоб замінити слова з помилками правопису.

Метою побудови N-gram моделей є визначення ймовірності вживання заданої фрази[4]. Цю ймовірність можна задати формально як ймовірність виникнення послідовності слів в якомусь наборі текстів. Наприклад, ймовірність фрази “щастя є задоволенням без каяття” можна обчислити як добуток ймовірностей кожного з слів цієї фрази (рис. 1.1).

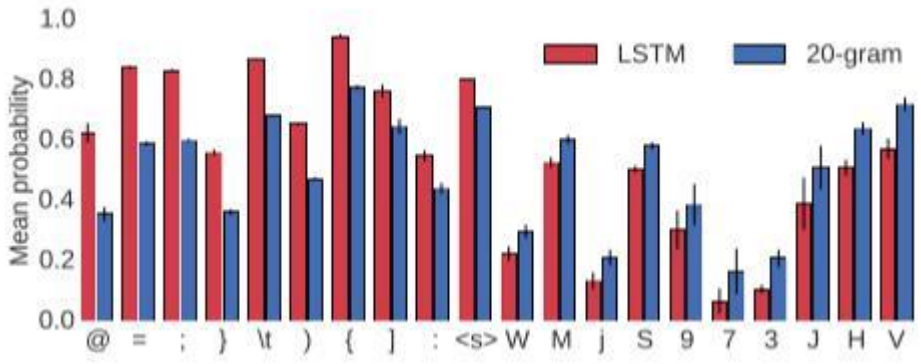

Рисунок 1.1 - Ймовірнісний розподіл N-грам моделі

В результаті, якщо ми порахуємо всі пари слів в деякому тексті, ми зможемо обчислити вірогідність довільної фрази. Цей набір розрахованих ймовірностей і буде N-gram моделлю.

1.1.3 Експоненціальні мовні моделі

Максимальні мовні моделі ентропії кодують зв'язок між словом і історією n-gram (рис. 1.2), використовуючи характеристичні функції (1.4)

$$P(w_1, \dots, w_m) = \frac{1}{Z(w_1, \dots, w_{m-1})} \exp(a^T f(w_1, \dots, w_m)), \quad (1.4)$$

де $Z(w_1, \dots, w_{m-1})$ - функція розділу,

a - параметризований вектор,

$f(w_1, \dots, w_m)$ - характеристична функція.

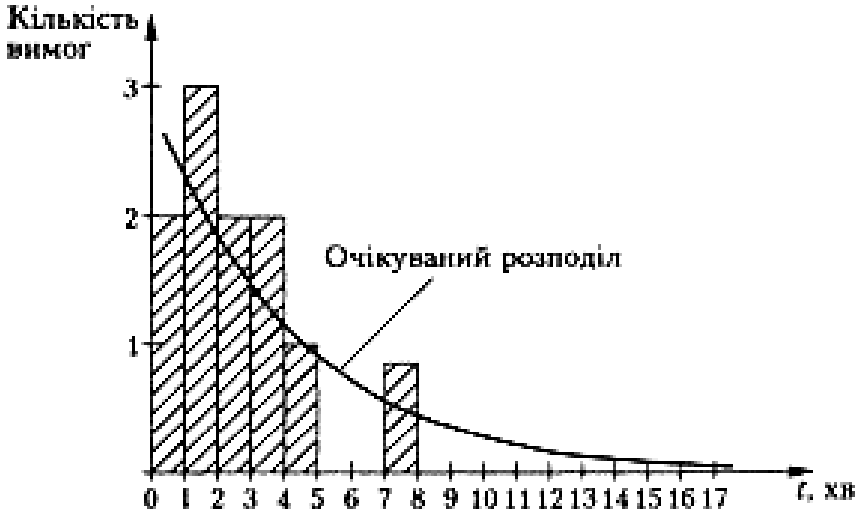

Рисунок 1.2 - Залежність кількості вимог від часу

У найпростішому випадку характеристична функція є лише індикатором наявності певної n-gram моделі. Корисно використовувати попередню версію на параметрі “a” або деяку форму регуляризації.

1.1.4 Нейромережні моделі

Моделі нейромереж використовують безперервні уявлення або вкладені слова для їх прогнозування[5]. Ці моделі використовують нейронні мережі (рис.1.3).

Оскільки мовні моделі навчаються на великих текстах, кількість унікальних слів збільшується, а кількість можливих послідовностей слів

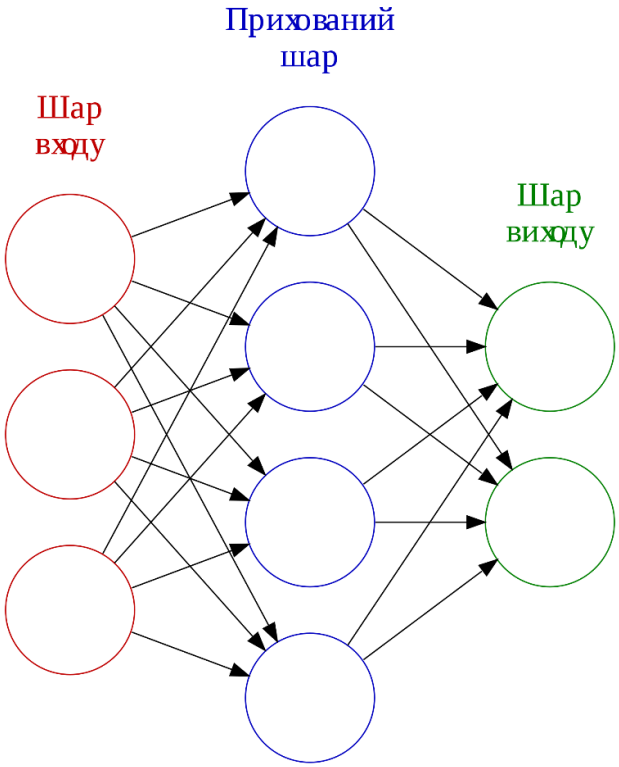

Рисунок 1.3 - Модель нейронної мережі

зростає експоненційно розміру словникового запасу, що спричиняє проблему ранжування даних. Таким чином, потрібна статистика для правильної оцінки ймовірностей. Нейромережі уникають цієї проблеми, представляючи слова в розподіленому вигляді, як нелінійні комбінації ваг у нейронній мережі. Альтернативний опис полягає в тому, що нейронна мережа наближає мовну функцію. Нейронна мережева архітектура може бути послідовною або періодичною.

Як правило, моделі нейромережевих мов будуються та навчаються як імовірнісні класифікатори, які навчаються прогнозувати розподіл ймовірностей (1.5)

$$P(w_t|context)\forall t \in V\ . \tag{1.5}$$

Мережа навчається передбачати розподіл ймовірності у словнику, якщо враховувати якийсь мовний контекст. Це робиться за допомогою стандартних алгоритмів тренування нейронних мереж, таких як стохастичний градієнтний спуск із зворотним поширенням. Контекст може бути вікном з попередніми словами з фіксованим розміром, так що прогнозує мережа (1.6) від вектору

$$P(w_t|w_{t-k}, \dots, w_{t-1}) \tag{1.6}$$

ознак, що представляє попередні слова. Іншим варіантом є використання "майбутніх" слів, а також "минулих" слів як функцій, так що оцінка імовірності (1.7)

$$P(w_t|w_{t-k}, \dots, w_{t-1}, w_{t+1}, \dots, w_{t+k}). \tag{1.7}$$

Третій варіант (1.8), який дозволяє швидше тренуватися, полягає в тому, щоб інвертувати попередню проблему і змусити нейронну мережу вивчати контекст, даючи слово

$$\sum_{-k\le j-1, j\le k} \log P(w_{t+j}|w_t)\ . \tag{1.8}$$

Це називається мовною моделлю skip-gram і є основою популярної програми “word2vec”.

Замість того, щоб використовувати нейронну мережеву мовну модель для створення реальних імовірностей, загальним замість цього є

використання розподіленого представлення, закодованого в "схованих" шарах мереж, як подання слів; кожне слово потім перетворюється на n-мірний реальний вектор, що називається вкладенням слова, де n - розмір шару безпосередньо перед вихідним шаром. Представлення в моделях скіп-грама мають чітку характеристику, що вони моделюють семантичні відносини між словами як лінійні комбінації, приймаючи форму композиції[6].

1.2 Огляд програмних засобів для проведення паралельних обчислень

1.2.1 NVIDIA CUDA

CUDA (Compute Unified Device Architecture) – програмно-апаратна архітектура паралельних обчислень, що дозволяє істотно збільшити обчислювальну продуктивність завдяки використанню графічних процесорів фірми NVIDIA[7].

Графічна підсистема комп'ютера з підтримкою CUDA може бути використана, як обчислювальна.

CUDA SDK (Software Development Kit) дозволяє програмістам реалізовувати на спеціальному спрощеному діалекті мови програмування C алгоритми, що виконуватимуться на графічних процесорах NVIDIA, і включати їх виклики в текст програми на C/C++. Архітектура CUDA (рис. 1.4) дає розробнику можливість на свій розсуд організовувати доступ до набору інструкцій графічного прискорювача і управляти його пам'яттю.

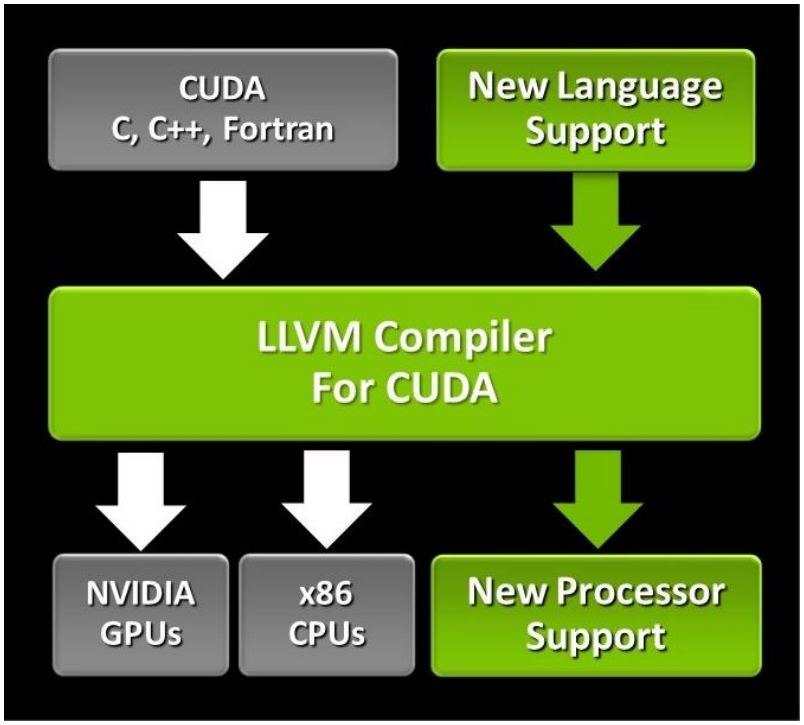

Рисунок 1.4 - Архітектура CUDA SDK

У порівнянні з традиційним підходом до організації обчислень загального призначення допомогою можливостей графічних API, у архітектури CUDA відзначають наступні переваги в цій області:

- інтерфейс програмування додатків CUDA (CUDA API) заснований на стандартному мові програмування C з деякими обмеженнями;
- спільна між потоками пам'ять (shared memory) розміром в 16 Кб може бути використана під організований користувачем кеш з більш широкою смугою пропускання, ніж при вибірці з звичайних текстур;
- більш ефективні транзакції між пам'яттю центрального процесора і відеопам'яттю;
- повна апаратна підтримка цілочисельних і побітових операцій;
- підтримка компіляції GPU коду засобами відкритого LLVM;
- підтримка рекурсивних викликів.

До недоліків вищеописаної архітектури можна віднести те, що виконувати, відлагоджувати та профілювати додатки, що були розроблені з використанням архітектури CUDA, можна лише на графічних процесорах фірми NVIDIA.

1.2.2 CNTK

The Microsoft Cognitive Toolkit (CNTK)[8] - це інструментарій з відкритим кодом для комерційного використання. Він описує нейронні мережі як серію обчислювальних кроків через спрямований граф. CNTK дозволяє користувачеві легко реалізувати та об'єднувати популярні типи моделей, такі як DNN-канали, згорткові нейронні мережі (CNN) та рекурентні нейронні мережі (RNN / LSTM) (рис. 1.5). CNTK реалізує стохастичний градієнтний спуск (SGD, зворотне поширення помилок) з автоматичною диференціацією та розпаралелюванням на кількох графічних процесорах та серверах.

CNTK може бути включений як бібліотека в програмах мовами Python, C# або C++ або використовуватися як окремий інструмент для машинного навчання за допомогою власної мови опису моделі (BrainScript).

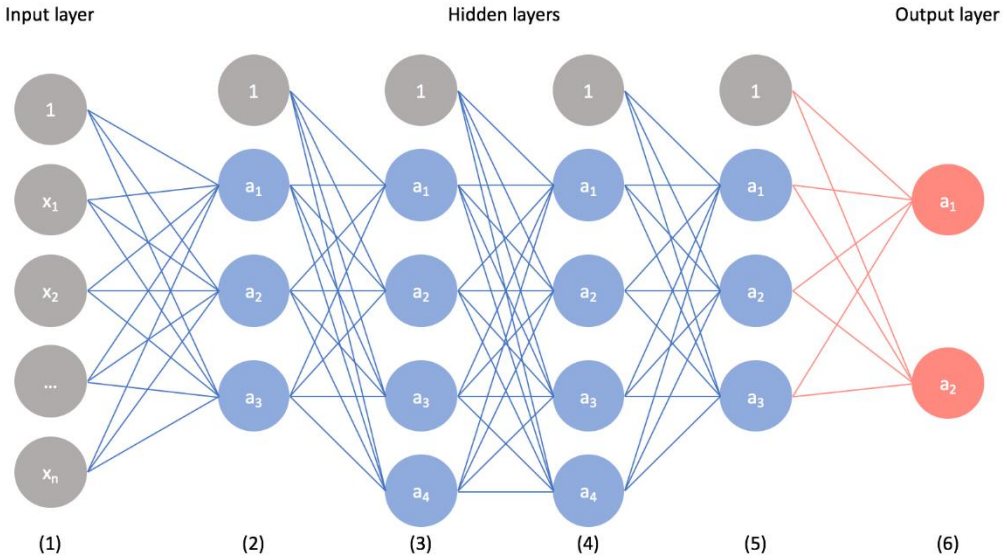

Рисунок 1.5 - Архітектура моделі LSTM мережі

CNTK Python API складається з абстракцій для визначення моделі та обчислень, алгоритмів навчання, читання даних та розподіленого навчання:

- гнучкість і компактність - ці абстракції дають змогу гнучко і стисло при визначати та тренувати довільні нейронні мережі;
- ефективні інтерфейси даних - прості, але легкі інтерфейси даних дозволяють користувачам ефективно передавати дані у вигляді нативних масивів мови пайтон у обчислювальне ядро;
- вбудовані зчитувачі даних - вбудовані оптимізовані та масштабовані читачі даних CNTK для зображень, текстових форматів також доступні для полегшення безпосереднього навчання з існуючими даними без потреби копіювати коди даних;
- високомасштабне навчання - API надає високомасштабні можливості розподіленої підготовки CNTK (алгоритми розпаралелювання, такі як 1-Bit SGD);
- короткий опис мережі - API включає в себе бібліотеку високорівневих

шарів, що дозволяє короткочасно визначити розширені визначення нейронних мереж, включаючи повторення, подібні до CNTK V1. Інструментарій підтримує представлення повторюваних моделей у символічній формі як цикли в нейронній мережі, замість того, щоб вимагати статичного розгортання кроків повторення. Це дає набагато більш загальне, лаконічне та ефективне представлення та виконання рекурентних нейронних мереж.

Всі основні обчислення, навчання та читання даних абстракцій CNTK Python API дуже легко розширюються як з Python, так і з C ++, що дозволяє користувачам впровадження нових операторів, учнів та читачів даних, які вільно створюються за допомогою вбудованих засобів бібліотеки.

API вводить новий формат серіалізації на базі протоколів Buffers, який підтримує зворотню сумісність збережених моделей.

1.2.3 TensorFlow

TensorFlow - це фреймворк програмного забезпечення з відкритим кодом для чисельних обчислень з високою ефективністю. Його гнучка архітектура (рис. 1.6) дозволяє легко розгортати обчислення на різних платформах

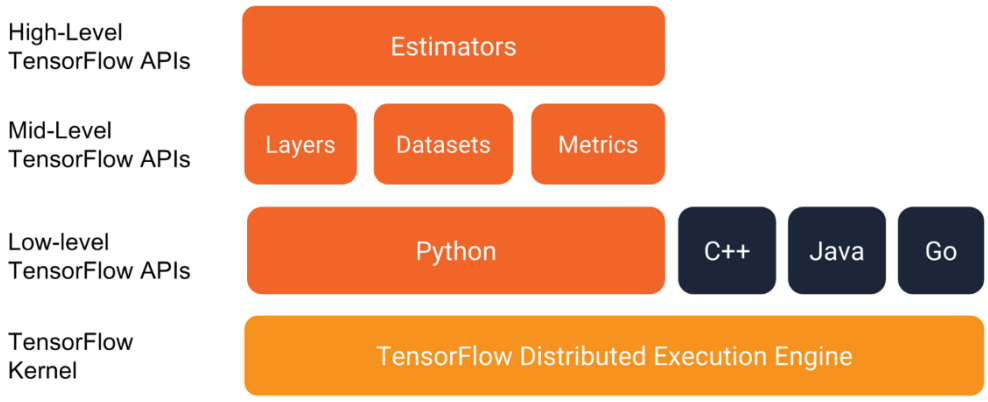

Рисунок 1.6 - Архітектура фреймворку TensorFlow

(процесори, графічні процесори, TPU), а також від настільних комп'ютерів до кластерів серверів, для мобільних і вбудованих пристроїв[8]. Розроблена дослідниками та інженерами команди “Google Brain” в організації “AI Google”, вона має сильну підтримку для машинного навчання та глибокого навчання, а гнучкі числові обчислення використовуються в багатьох інших наукових областях.

1.2.4 Вибір технології ведення паралельних обчислень

Порівнюючи всі недоліки вищезгаданих технологій для вирішення поставленої задачі було обрано фреймворк TensorFlow за наступні його переваги перед його конкурентами. Одночасно з простим використанням TensorFlow розробники можуть використовувати TensorFlow на нових мовах. TensorFlow.js - це нова схема ML для розробників JavaScript. Автоматичне навчання в браузері за допомогою TensorFlow.js відкриває нові можливості, включаючи інтерактивний ML та підтримку сценаріїв, де всі дані залишаються на стороні клієнта. Він може бути використаний для побудови і підготовки модулів цілком у веб-переглядачі, а також імпортувати моделі TensorFlow та Keras навчених в автономному режимі для подальшої роботи з використанням прискорення WebGL. Також регулярно виходять оновленнями до TensorFlow Lite, легкого, між платформного рішення TensorFlow для розгортання навчених моделей ML на мобільних пристроях та інших пристроях. Окрім існуючої підтримки для Android та iOS, додалась підтримка для Raspberry Pi. Основний інтерпретатор TensorFlow Lite зараз складає лише 75 Кб (порівняно з 1,1 Мб для TensorFlow), і спостерігається прискорення до 3-х разів при використанні квантування моделей зображень зображень на TensorFlow Lite проти TensorFlow. Для апаратної підтримки

TensorFlow має інтеграцію з TensorRT NVIDIA. TensorRT[9] - це бібліотека, яка оптимізує глибокі моделі навчання для висновку та створює середу виконання для розгортання на GPU у виробничих середовищах. Це дає ряд оптимізацій TensorFlow і автоматично вибирає ядро платформи, щоб максимально збільшити пропускну здатність і мінімізувати затримку на обчислення висновку на графічних процесорах. Для запуску TensorFlow на процесорах, партнерство з Intel забезпечило інтеграцію з високо оптимізованою бібліотекою з відкритим кодом Intel MKL-DNN для глибокого вивчення. При використанні Intel MKL-DNN, ми спостерігали прискорення висновків до 3-х разів на різних платформах Intel CPU. Список платформ, на яких можливо запустити TensorFlow, виросло на Cloud TPU, які були випущені бета-версією минулого місяця. З моменту запуску команда Google Cloud TPU вже досягла значного підвищення продуктивності 1,6X у продуктивності ResNet-50. Ці поліпшення будуть доступні для користувачів TensorFlow з версією 1.8 незабаром.

2. РОЗРОБКА МЕТОДУ ПРИСКОРЕННЯ СТАТИСТИЧНОГО МОДЕЛЮВАННЯ МОВ НА ОСНОВІ ВИКОРИСТАННЯ ФРЕЙМВОРКУ TENSORFLOW

Механізми уваги в нейронних мережах[10] базуються на механізмі візуальної уваги, виявленого у людей. Зорова увага людини добре вивчена, і в той час як існують різні моделі, всі вони, по суті, орієнтуються на певну область зображення з "високою роздільною здатністю" при сприйнятті навколишнього зображення в "низькій роздільній здатності", а потім коригується фокусна відстань з часом. Увага в нейронних мережах має довгу історію, особливо в області розпізнавання образів. Але лише останнім часом механізми уваги перетворилися на повторювані архітектури нейронних мереж, які зазвичай використовуються в NLP[11] (і все частіше також у комп’ютерному зорі).

2.1 Адаптування механізму уваги рекурентної нейромережі для використання у фреймворку TensorFlow

2.1.1 Загальний опис механізму уваги

Щоб зрозуміти, що може зробити для нас механізм уваги, використаємо приклад нейромережевого перекладу (NMT). Системи традиційного машинного перекладу, як правило, покладаються на складну функціональність на основі статистичних властивостей тексту. Ці системи є складними, і в них вкладається багато інженерних зусиль. Системи нейромережевого перекладу працюють трохи по-іншому. У NMT ми перетворюємо значення речення на векторне представлення фіксованої

довжини, а потім генеруємо переклад на основі цього вектора. Не спираючись на речі, як на псевдограми, а намагаючись зафіксувати сенс тексту вищого рівня, системи NMT узагальнюють нові речення краще, ніж багато інших підходів. Що більш важливо, системи NTM набагато простіше побудувати та тренувати, і вони не вимагають будь-якої ручної інженерії.

Більшість систем NMT працюють шляхом кодування вихідного речення (наприклад, німецьке речення) у вектор, що використовує рекурентну нейронну мережу, а потім декодування англійської пропозиції на основі цього вектора, також використовуючи RNN[11].

Слова "Echt", "Dicke" і "Kiste" потрапляють в кодер, і після спеціального сигналу декодер починає генерувати перекладену пропозицію. Декодер продовжує генерувати слова, доки не з'явиться спеціальний токен кінця речення. Тут вектори h представляють внутрішній стан кодера (рис. 2.2).

Декодер повинен генерувати переклад виключно на основі останнього прихованого стану (h_3 вище) від кодера. Цей вектор h3 повинен кодувати все, що нам потрібно знати про вихідне речення. Він повинен повністю зафіксувати його значення. У більш технічному плані цей вектор є вкладеним речення. Насправді, якщо ви розмістите вклади різних речень в маломірному просторі, використовуючи PCA або t-SNE для зменшення розміру, ви можете побачити, що семантично схожі фрази наближаються один до одного.

Ми можемо кодувати всю інформацію при потенційно дуже довгому реченні в єдиний вектор, а потім декодер створює хороший переклад на основі лише цього. Наприклад, вихідне речення становить 50 слів. Перше слово англійського перекладу, мабуть, дуже співвідноситься з першим словом вихідного речення. Але це означає, що декодер повинен враховувати інформацію від 50 кроків тому, і ця інформація повинна бути якось закодованою у векторі. Як відомо, періодичні нейронні мережі мають

проблеми з такими далекими залежностями. У теорії архітектури, подібні до LSTM[12], повинні мати справу з цим, але на практиці довготривалі залежності все ще залишаються проблематичними. Повернення вихідної послідовності (подача його назад в кодер) робить значно кращі результати, оскільки скорочує шлях від декодера до відповідних частин кодувача. Подібним чином, подання послідовності вводу двічі також допомагає мережі краще запам'ятовувати речі.

За допомогою механізму уваги ми більше не намагаємося закодувати повне вихідне речення у векторі фіксованої довжини. Швидше за все, декодеру дозволяється "відвідувати" різні частини початкового речення на кожному кроці вихідного кодування. Важливо, що моделі дозволяється дізнатися, що брати до уваги, виходячи з вхідного речення та того, що він зробив до цих пір. Отже, на мовах, які досить добре вирівняні (наприклад, англійською та німецькою мовами), декодер вирішив послідовно відвідувати речі. Приймаючи перше слово при виготовленні першого англійського слова тощо. Ось що було зроблено в “нейтральному машинному перекладі, спільно навчившись вирівнювати і перекладати” і виглядає так (рис. 2.2).

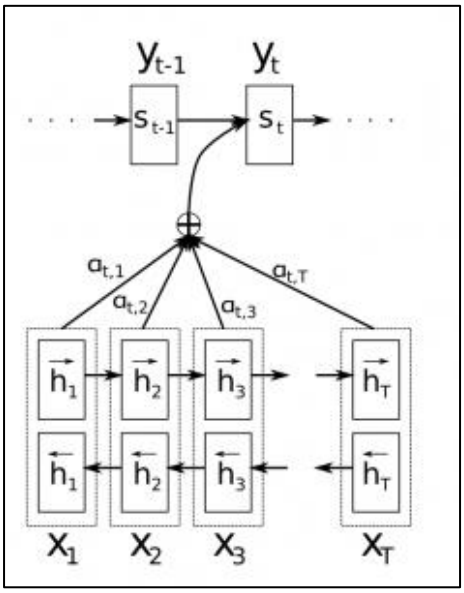

Рисунок 2.2 - Нейтральний машинний переклад

Тут “y” є нашими перекладеними словами, вироблені декодером, і “x” є нашими вихідними словами речення. Наведена вище ілюстрація використовує двонаправлену рекурентну мережу, але це не важливо, і ви можете просто ігнорувати зворотній напрямок. Важлива частина полягає в тому, що кожен вихідний код декодера y_t тепер залежить від вагової комбінації всіх вхідних станів, а не тільки останнього стану. A є ваги, які визначають, якою мірою кожний вихідний стан повинен розглядатися для кожного виходу. Отже, якщо a_t (3,2) - велике число, це означає, що декодер приділяє велику увагу другому стану в початковому реченні під час створення третього слова цільового речення. Стандарти “a”, як правило, нормалізуються до суми 1 (вони є розподілом над вхідними станами).

Якщо ми подивимося трохи більше на рівняння для уваги, ми можемо побачити, що увага прирівнюється до вартості. Нам потрібно підрахувати значення уваги для кожної комбінації вхідного і вихідного слова. Якщо у вас є 50-слівська вхідна послідовність і генерується 50 слів вихідної послідовності, буде 2500 значень уваги. Це не так вже й погано, але якщо ви проводите обчислення на рівні персонажів і використовуєте послідовності, що складаються з сотень токенів, вищезазначені механізми уваги можуть стати надмірно дорогими[13].

Насправді, це абсолютно протилежне. Людська увага - це те, що передбачає збереження обчислювальних ресурсів. Зосереджуючись на одній справі, ми можемо нехтувати багатьма іншими речами. Але це не те, що ми робимо в наведеній вище моделі. По суті, ми розглядаємо все докладно, перш ніж приймати рішення про те, на чому зосередитися. Інтуїтивно цей еквівалент виводить перекладене слово, а потім повертається через усю внутрішню пам'ять тексту, щоб вирішити, яке слово слід згенерувати далі. Це здається сміливим, але не зовсім тим, що роблять люди. Фактично, це більше

схоже на доступ до пам'яті, а не на увагу, що, на мій погляд, є дещо неправильним. Тим не менш, це не зупинило механізми уваги, щоб стати досить популярними і добре працювати над багатьма завданнями.

Альтернативним підходом до уваги є використання "підсилення навчання", щоб прогнозувати приблизне місцезнаходження, до якого потрібно зосередитись. Це звучить набагато більше схоже на людську увагу, і саме це зроблено в "рекурентних моделях" "візуальної уваги".

До цих пір ми звернули увагу на увагу на прикладі машинного перекладу. Але такий самий механізм уваги може бути застосований до будь-якої рекурентної моделі. Тож давайте подивимося ще на кілька прикладів.

У програмі "Шоу", "Відвідування" та "Скажіть" автори застосовують механізми уваги до проблеми створення опису зображень. Вони використовують Convolutional Neural Network для кодування зображення, а також повторювану нейронну мережу з механізмами уваги для створення опису. Візуалізуючи ваги уваги (як і в прикладі перекладу), ми інтерпретуємо те, на що модель дивиться під час генерації слова (рис. 2.3).

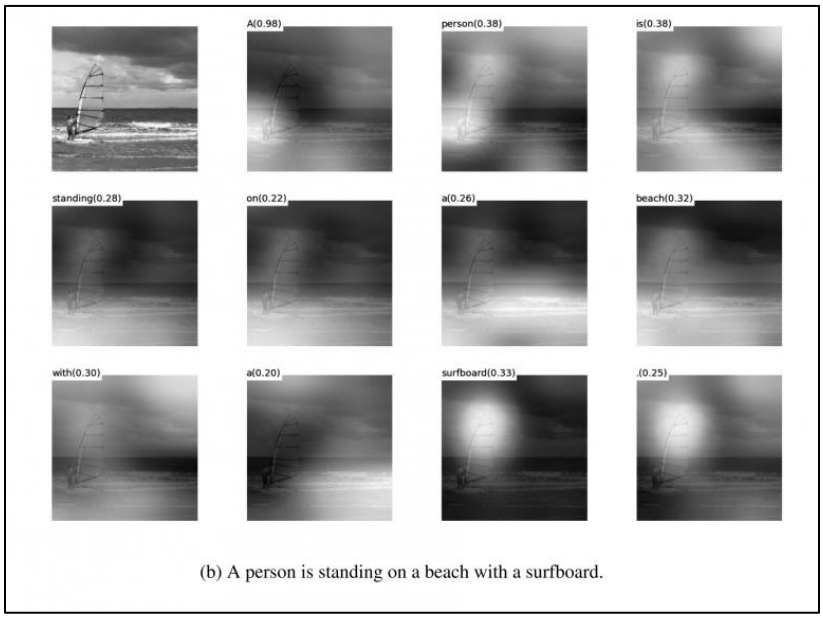

Рисунок 2.3 - Візуалізуючі ваги уваги

У граматиці іноземної мови автори використовують рекурентну нейронну мережу з механізмом уваги, щоб генерувати дерева розбору слів. Візуалізована матриця уваги дає уявлення про те, як мережа генерує ті дерева (рис. 2.4):

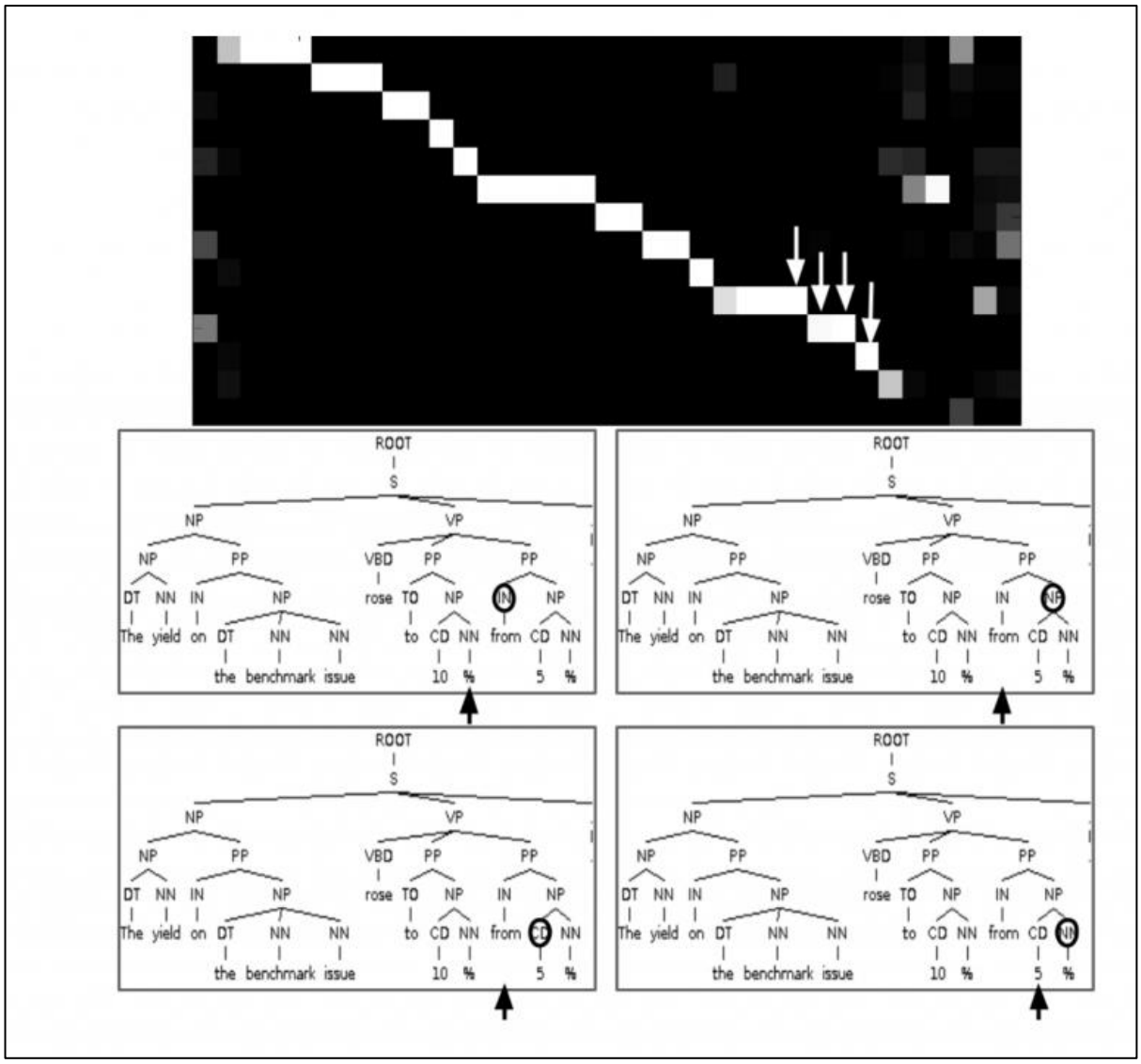

Рисунок 2.4 - Візуалізована матриця уваги

При навчанні машин для читання та розуміння автори використовують RNN для читання тексту, читання (синтетично сформованого) запитання, а потім дають відповідь. Візуалізуючи матрицю уваги (рис. 2.4), ми можемо

бачити, на чому мережі зосереджуються, коли намагаються знайти відповідь на запитання (рис. 2.5).

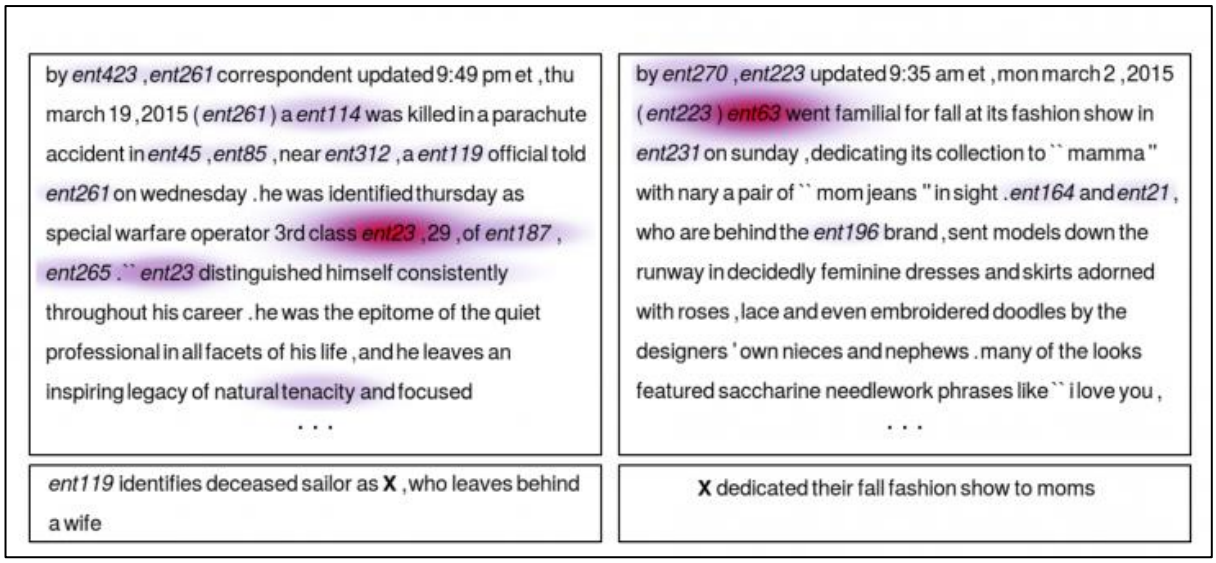

Рисунок 2.5 - Візуалізація роботи механізму уваги

2.1.2 Адаптування механізму уваги

Нижче я описую приклад механізму уваги, який використовувався в ряді сучасних систем, включаючи додатки з відкритим кодом, таких як OpenNMT та API TF seq2seq у його документації. Я також забезпечив зв'язок з іншими варіантами механізму уваги[13].

Метод уваги - приклад системи уваги NMT. Я докладно виділяю перший крок обчислення уваги. Для ясності я не показую вкладені та проекційні шари на малюнку.

Обчислення уваги відбувається на кожному етапі часу декодера. Він складається з наступних етапів (рис. 2.6).

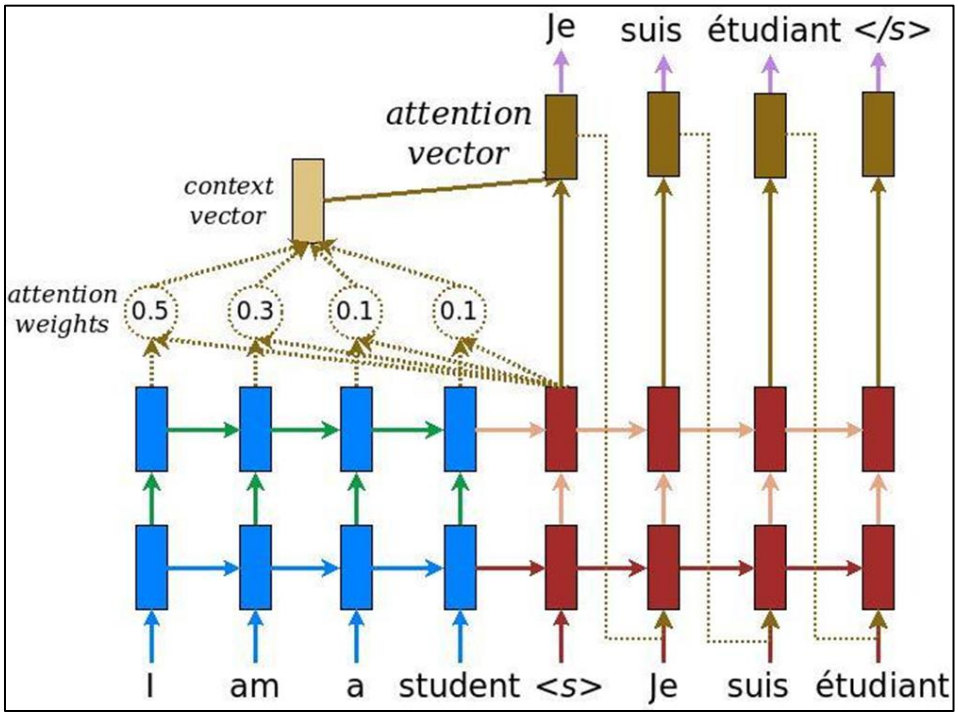

Рисунок 2.6 - Етапи роботи механізму уваги

Поточний цільовий прихований стан порівнюється з усіма джерелами, щоб отримати від уваги ваги (2.1).

Виходячи з ваг уваги, ми обчислюємо вектор контексту як середньозважене значення вихідних станів (2.2).

Об'єднайте вектор контексту з поточним цільовим прихованим станом, щоб отримати остаточний вектор уваги (2.3).

Вектор уваги подають як вхід до наступного етапу (подання вводу). Перші три етапи можна підсумувати за рівняннями нижче[14]:

$$a_{ts} = \frac{exp(score(h_t, \overline{h_s}))}{\sum_{\acute{s}=1}^{S} exp(score(h_t, \overline{h_{\acute{s}}}))} \quad (2.1)$$

$$c_t = \sum_s a_{ts} \bar{h}_s \quad (2.2)$$

$$a_t = f(c_t, h_t) = tanh(W_c[c_t; h_t]) \quad (2.3)$$

Тут функціональна оцінка використовується для порівняння цільового прихованого стану з кожним із прихованих станів джерела, а результат нормується на отриману увагу (розподіл по позиціях джерела). Є різні варіанти функції скорингу. Популярні забивні функції включають мультиплікативні і адитивні форми, наведені в рівнянні. Після обчислення вектору уваги використовується для виявлення значення softmax та втрати. Це схоже на цільовий прихований стан на верхньому шарі моделі vanilla seq2seq. Функція f може також приймати інші форми.

2.2. Адаптування механізму пам’яті рекурентної нейромережі для використання у фреймворку TensorFlow

Основна проблема, яку вирішує механізм уваги, полягає в тому, що він дозволяє мережі подивитися назад до вхідної послідовності, замість того, щоб змушувати робити кодування всієї інформації в одному векторі фіксованої довжини. Як я вже згадував вище, я думаю, що увага є дещо неправильною.

2.2.1. Загальний опис механізму пам’яті

Інший варіант інтерпретування механізму уваги полягає в тому що він просто дає мережевий доступ до своєї внутрішньої пам'яті, яка є прихованим станом кодувача. У цій інтерпретації, замість того, щоб вибрати, до чого "брати участь", мережа вибирає те, що потрібно витягнути з пам'яті. На відміну від типової пам'яті, механізм доступу до пам'яті тут є м'яким, що означає, що мережа отримує зважену комбінацію всіх розташувань пам'яті, а не значення з одного дискретного розташування. М’яке отримання доступу до пам'яті має вигоду в тому, що ми можемо легко тренувати мережу повноцінним способом, використовуючи зворотне поширення (хоча існують інші підходи, за яких градієнти обчислюються методами вибірки замість зворотного розповсюдження).

Самі механізми пам'яті мають набагато довшу історію. Прихований стан стандартної рекурентної нейронної мережі сам по собі є типом внутрішньої пам'яті. RNN страждають від втрати градієнта, що перешкоджає їм вивчати довготривалі залежності. На фоні цьому LSTM виглядає краще, використовуючи механізм стримування, який дозволяє видаляти і оновлювати явні області пам'яті[15].

Тенденція до більш складних структур пам'яті зараз триває. "End-To-End" мережі дозволяють мережі кілька разів прочитати ту ж вхідну послідовність, перш ніж робити генерацію, оновлюючи вміст пам'яті на кожному кроці. Наприклад, відповідаючи на запитання, зробивши декілька розумних кроків над матеріалом вводу. Однак, коли певні ваги параметрів мережі прив'язані певним чином, механізм пам'яті в "End-to-End" мережах ідентичний механізму уваги, представленому тут, лише тим, що він здійснює

декілька операцій над пам'яттю (оскільки він намагається інтегрувати інформацію з кількох речень).

Машини нейромережі Тьюрінга використовують подібну форму механізму пам'яті, але з більш складним типом адресації, яка використовує як на основі вмісту (як тут) так і адресну адресацію, що дозволяє мережі вивчати шаблон адресації для виконання простих комп'ютерних програм, таких як алгоритми сортування.

Цілком імовірно, що в майбутньому ми побачимо більш чітку відмінність між механізмами пам'яті та увагою, можливо, у відповідності до зростаючого інтересу до навчання Neural Turing Machines, які намагаються вивчати схеми доступу до пам’яті для представлення зовнішніх інтерфейсів.

2.2.2. Теоретична оцінка покращення якості та швидкодії процесу тренування та роботи статичного моделювання мови

Ключовою ідеєю механізму уваги є встановлення прямих короткострокових зв'язків між ціллю та джерелом, звертаючи увагу на відповідний вихідний вміст під час перекладу. Хороший побічний продукт механізму уваги – це можливість легко візуалізувати матрицю вирівнювання між вихідними і цільовими пропозиціями (рис. 2.7).

У моделі vanilla seq2seq я передаю останній стан джерела від датчика до декодера на початку процес декодування. Це добре працює для коротких та середніх передбачень. Однак, для довгих передбачень, одна прихована фіксована величина стає інформаційно вузьким місцем. Замість того, щоб відкинути всі приховані стани, обчислені в джерелі RNN, механізм уваги забезпечує підхід, який дозволяє декодеру заглянути в них (розглядаючи їх як динамічну пам'ять вихідної інформації).

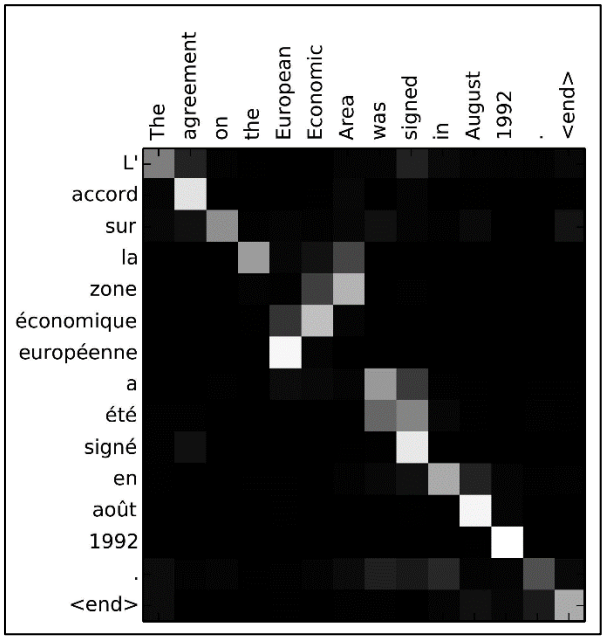

Рисунок 2.7 - Вирівнювання між джерелом і цільовими реченнями

Таким чином, механізм уваги покращує переклад довгих речень. В даний час механізми уваги є стандартом defacto і успішно застосований до багатьох інших завдань (у тому числі створення заголовків зображення, розпізнавання мовлення та підведення підсумків тексту).

2.3. Передбачення наступного слова у тексті маючи попередній текст на основі використання фреймворку TensorFlow

Мета завдання полягає в тому, щоб відповідати імовірнісній моделі, яка присвоює ймовірності реченням. Це відбувається, передбачаючи наступні слова в тексті, що дає історію попередніх слів. Для цього я використав набір даних Penn Tree Bank (PTB), який є популярним еталоном для вимірювання якості цих моделей, хоча він малий і відносно швидкий для тренувань[16].

Набір даних вже підготовлений і містить у загальному 10000 різних слів, включаючи маркер кінця речення, і спеціальний символ (<УНК>) для

рідкісних слів. У reader.py я перетворюю кожне слово на унікальний цільний ідентифікатор, щоб нейронна мережа змогла легко обробляти дані.

Ядро моделі складається з клітини LSTM, яка одночасно обробляє одне слово і обчислює імовірності можливих значень для наступного слова у

```
 t=0  t=1    t=2  t=3     t=4
[The, brown, fox, is,     quick]
[The, red,   fox, jumped, high]

words_in_dataset[0] = [The, The]
words_in_dataset[1] = [brown, red]
words_in_dataset[2] = [fox, fox]
words_in_dataset[3] = [is, jumped]
words_in_dataset[4] = [quick, high]
batch_size = 2, time_steps = 5
```

Рисунок 2.8 - Ядро моделі з клітини LSTM

реченні. Стан пам'яті мережі ініціалізується вектором нулів і оновлюється після прочитання кожного слова. З обчислювальних причин я обробляв дані в міні-партіях розміром batch_size (рис. 2.8).

```
words_in_dataset = tf.placeholder(tf.float32, [time_steps, batch_size, num_features])
lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)
# Initial state of the LSTM memory.
hidden_state = tf.zeros([batch_size, lstm.state_size])
current_state = tf.zeros([batch_size, lstm.state_size])
state = hidden_state, current_state
probabilities = []
loss = 0.0
for current_batch_of_words in words_in_dataset:
    # The value of state is updated after processing each batch of words.
    output, state = lstm(current_batch_of_words, state)

    # The LSTM output can be used to make next word predictions
    logits = tf.matmul(output, softmax_w) + softmax_b
    probabilities.append(tf.nn.softmax(logits))
    loss += loss_function(probabilities, target_words)
```

Рисунок 2.9 - Псевдокод програми обробки наступного слова

Важливо зазначити, що current_batch_of_words не відповідає "реченню" слів (рис. 2.9.). Кожне слово в партії відповідає часу t. TensorFlow автоматично

сумує градієнти кожної партії. За конструкцією вихід нейронної мережі (RNN) залежить від довільно віддалених входів. На жаль, це робить обчислення зворотнього розповсюдження важким. Для того, щоб зробити навчальний процес тривимірним, загальноприйнятою практикою є створення "розгорнутої" версії мережі, яка містить фіксований номер (num_steps) входів та виходів LSTM. Потім модель навчається за цим кінцевим наближенням RNN. Це може бути здійснено шляхом подачі входів довжини num_steps за один раз і виконання зворотнього проходу після кожного такого вхідного блоку.

Ось спрощений блок коду для створення графа, який виконує усічене зворотнє поширення (рис. 2.10).

```
# Placeholder for the inputs in a given iteration.
words = tf.placeholder(tf.int32, [batch_size, num_steps])

lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)
# Initial state of the LSTM memory.
initial_state = state = tf.zeros([batch_size, lstm.state_size])

for i in range(num_steps):
    # The value of state is updated after processing each batch of words.
    output, state = lstm(words[:, i], state)

    # The rest of the code.
    # ...

final_state = state
```

Рисунок 2.10 - Створення графа, який виконує усічене зворотнє поширення

це як здійснити ітерацію по всьому набору даних (рис. 2.11).

```
# A numpy array holding the state of LSTM after each batch of words.
numpy_state = initial_state.eval()
total_loss = 0.0
for current_batch_of_words in words_in_dataset:
    numpy_state, current_loss = session.run([final_state, loss],
        # Initialize the LSTM state from the previous iteration.
        feed_dict={initial_state: numpy_state, words: current_batch_of_words})
    total_loss += current_loss
```

Рисунок 2.11 - Ітерація по всьому набору даних

Ідентифікатори слів будуть вбудовані в щільне подання перед подачею до LSTM. Це дозволяє моделі ефективно представляти знання про конкретні слова. Також це легко реалізувати (рис. 2.12).

```
# embedding_matrix is a tensor of shape [vocabulary_size, embedding size]
word_embeddings = tf.nn.embedding_lookup(embedding_matrix, word_ids)
```

Рисунок 2.12 - Ущільнення вхідних даних перед подачею до LSTM

Матриця вкладок буде ініціалізована випадковим чином, і модель навчиться диференціювати значення слів, просто переглянувши дані.

Я звів до мінімуму вірогідність середнього негативного журналу цільових слів (2.4):

$$loss = -\frac{1}{N}\sum_{i=1}^{N} ln\, p_{target_i} \tag{2.4}$$

Це не дуже важко реалізувати, але функція sequence_loss_by_example вже доступна, тому я можемо просто використати її тут.

Типова міра, про яку повідомляється в документах - це усереднене значення для кожного слова (2.5):

$$e^{-\frac{1}{N}\sum_{i=1}^{N} ln\, p_{target_i}} = e^{loss} \quad (2.5)$$

І я буду стежити за його цінністю протягом усього тренувального процесу.

Щоб надати моделі виразнішу силу, ми можемо додати кілька рівнів LSTM для обробки даних. Вихід першого шару стане входом другого і так далі (рис. 2.13).

```
def lstm_cell():
  return tf.contrib.rnn.BasicLSTMCell(lstm_size)
stacked_lstm = tf.contrib.rnn.MultiRNNCell(
    [lstm_cell() for _ in range(number_of_layers)])

initial_state = state = stacked_lstm.zero_state(batch_size, tf.float32)
for i in range(num_steps):
    # The value of state is updated after processing each batch of words.
    output, state = stacked_lstm(words[:, i], state)

    # The rest of the code.
    # ...

final_state = state
```

Рисунок 2.13 - Псевдокод безперервної реалізації багаторівневого LSTM

У нас є клас під назвою MultiRNNCell, який робить реалізацію безперервною.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ МЕТОДУ ПРИСКОРЕННЯ СТАТИСТИЧНОГО МОДЕЛЮВАННЯ МОВ ПРОГРАМУВАННЯ НА ОСНОВІ ВИКОРИСТАННЯ ФРЕЙМВОРКУ TENSORFLOW

Мовна модель буде статистичною і передбачатиме ймовірність кожного слова, заданого вхідною послідовністю тексту. Прогнозоване слово буде подаватися як вхід, щоб, у свою чергу, генерувати наступне слово[17].

Ключовим параметром є тривалість вхідних послідовностей. Вони повинні бути достатньо довгими, щоб модель могла вивчати контекст для наступних слів, які вона буде передбачати. Довжина вхідного тексту також визначатиме довжину базового тексту, що використовується для створення нових послідовностей при використанні моделі.

3.1 Програмна реалізація модуля завантаження текстових даних

Для роботи з вхідним текстом потрібно реалізувати завантаження тексту в пам'ять. Я реалізував базову функцію, щоб завантажити весь текстовий файл у пам'ять і повернути його зміст з функції. Ця функція називається load_doc (рис. 3.1).

```
1   # load doc into memory
2   def load_doc(filename):
3       # open the file as read only
4       file = open(filename, 'r')
5       # read all text
6       text = file.read()
7       # close the file
8       file.close()
9       return text
10
```

Рисунок 3.1 - Завантаження файлу у пам'ять

Вказуючи ім'я файлу, вона повертає послідовність завантаженого тексту. Використовуючи цю функцію, я завантажую більш чистий варіант вхідного документа з будь якого файлу наступним чином (рис. 3.2).

```
1  # load document
2  in_filename = 'republic_clean.txt'
3  doc = load_doc(in_filename)
4  print(doc[:200])
```

Рисунок 3.2 - Зчитування даних з файлу

3.2 Програмна реалізація модуля обробки текстових даних

Для перетворення вихідного тексту у послідовність токенів або слів, які використовуватимуться як вхідні дані для навчання моделі потрібно зробити адаптування вхідних даних.

Після аналізу вихідного тексту нижче наведено деякі конкретні операції, які я виконуватиму для очищення тексту. У подальшому розширенні системи можуть додаватися додаткові операції очищення в якості розширень:

- Заміна дефіса на пробіл для кращого розділення слів;
- Розбиття слів на основі пробілу;
- Видалення всіх знаків пунктуації зі слів для зменшення розміру словника;
- Видалення всіх слів, які не є алфавітними, щоб видалити додаткові маркери пунктуації;
- Нормалізація всіх слів в нижньому регістрі для зменшення розміру словника.

Розмір словника – це важливий критерій оптимізації по часу тренування моделі. Менший словник призводить до меншої моделі, яка тренується

швидше.

Реалізовано кожну з цих операцій для нормалізації вхідного тексту в такому порядку за допомогою функції. Нижче наведено функцію clean_doc, яка завантажує документ як аргумент і повертає масив чистих токенів (рис. 3.3).

```
import string

# turn a doc into clean tokens
def clean_doc(doc):
    # replace '--' with a space ' '
    doc = doc.replace('--', ' ')
    # split into tokens by white space
    tokens = doc.split()
    # remove punctuation from each token
    table = str.maketrans('', '', string.punctuation)
    tokens = [w.translate(table) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # make lower case
    tokens = [word.lower() for word in tokens]
    return tokens
```

Рисунок 3.3 - Підготовка токенів

Запустивши цю операцію з очищення на завантаженому документі та роздрукувавши деякі токени та статистику можна побачити що функція відпрацьовує коректно.

По-перше, в результаті виклику функції нормалізації даних бачимо чистий список токенів, які виглядають структурованіше за сирий текст. Також стають доступними деякі статистичні дані про вхідний документ.

Проаналізувавши, що у вхідному тексті менше 120000 слів, а у словнику - трохи менше 7500 слів можна зробити висновок, що отриманий словник

невеликий, а моделі, які підходять для цих даних, повинні бути керованими на слабкому обладнанні[18].

3.3 Програмна реалізація розподіленого сховища набору даних для тренування

3.3.1 Реалізація структур даних, що проводять обчислення за ознаками

Потрібно організувати довгий список токенів у послідовності з 50 вхідних слів та 1 вихідного слова. Це можна зробити, повторюючи список токенів з токену 51 і давши попередні 50 токенів у вигляді послідовності, а потім повторити цей процес до кінця списку токенів[18].

Токени слід розбити на рядки, розділивши пробілом, для подальшого зберігання у файл (рис. 3.4).

```
# organize into sequences of tokens
length = 50 + 1
sequences = list()
for i in range(length, len(tokens)):
    # select sequence of tokens
    seq = tokens[i-length:i]
    # convert into a line
    line = ' '.join(seq)
    # store
    sequences.append(line)
print('Total Sequences: %d' % len(sequences))
```

Рисунок 3.4 - Розбиття списку чистих токенів на послідовності

Виконання цієї частини створює довгий список рядків.

Описавши статистику в списку, бачимо, що буде рівно 118,633 навчальних моделей, які відповідають нашим вимогам до моделі.

Далі послідовності зберігаються в новий файл для подальшого завантаження.

Я визначив додаткову функцію для збереження рядків тексту в файл. Ця нова функція називається save_doc і наведена вище. В якості вводу потрібен список рядків та ім'я файлу. Слова записуються по одному на рядок у форматі ASCII.

3.3.2 Розробка модуля для збереження значень ознак

Тепер модель статистичної мови може бути створена з підготовлених даних.

Модель, яку ми тренуємо, має кілька унікальних характеристик:

- Використовується розподілене представлення для слів, так що різні слова зі схожими значеннями матимуть аналогічне подання;
- Представлення вивчається одночасно з вивченням моделі;
- Прогнозування ймовірності наступного слова відбувається, використовуючи контекст останніх 100 слів.

Для отримання таких характеристик потрібно використовувати вбудований шар для вивчення представлення слів і довготривалої короткотермінової пам'яті (LSTM) повторюваної нейронної мережі, щоб навчитися передбачати слова на основі їх контексту.

Першим кроком буде завантаження навчальних даних за допомогою функції load_doc, яка розроблена в попередньому підрозділі.

Після завантаження дані потрібно розділити на окремі тренувальні послідовності шляхом розщеплення на основі символу нового рядку.

3.4 Програмна реалізація модуля тренування мовної моделі

Тепер ми можемо визначити та реалізувати тестову мовну модель на отриманих навчальних даних (рис. 3.5).

```
# define model
model = Sequential()
model.add(Embedding(vocab_size, 50, input_length=seq_length))
model.add(LSTM(100, return_sequences=True))
model.add(LSTM(100))
model.add(Dense(100, activation='relu'))
model.add(Dense(vocab_size, activation='softmax'))
print(model.summary())
```

Рисунок 3.5 - Тренування мовної моделі

Вивчаючи рівень вкладання потрібно враховувати розмір словника та довжину вхідних послідовностей, які обраховано раніше. Вкладення також має параметр, щоб вказати, скільки параметрів буде використано для відображення кожного слова. Тобто, розмір вбудованого векторного простору[19].

Загальні значення - 50, 100 і 300. Я використовую тут 50, але розгляну тестування менших або більших значень.

Я використав два прихованих шари LSTM з 100 комірками пам'яті кожен. Більше комірок пам'яті та глибша мережа можуть продемонструвати кращі результати.

Зовнішній повністю з'єднаний шар із 100 нейронів підключається до прихованих шарів LSTM, щоб інтерпретувати функції, витягнуті з послідовності. Вихідний шар передбачає наступне слово як окремий вектор,

з ймовірностями для кожного слова в словнику. Функція активації soft max використовується для того, щоб виходи мали характеристики нормалізованих імовірностей.

Кінцевий стан мережі друкується як перевірка коректності, щоб переконатись, що побудована мережа відповідає вимогам (рис. 3.6).

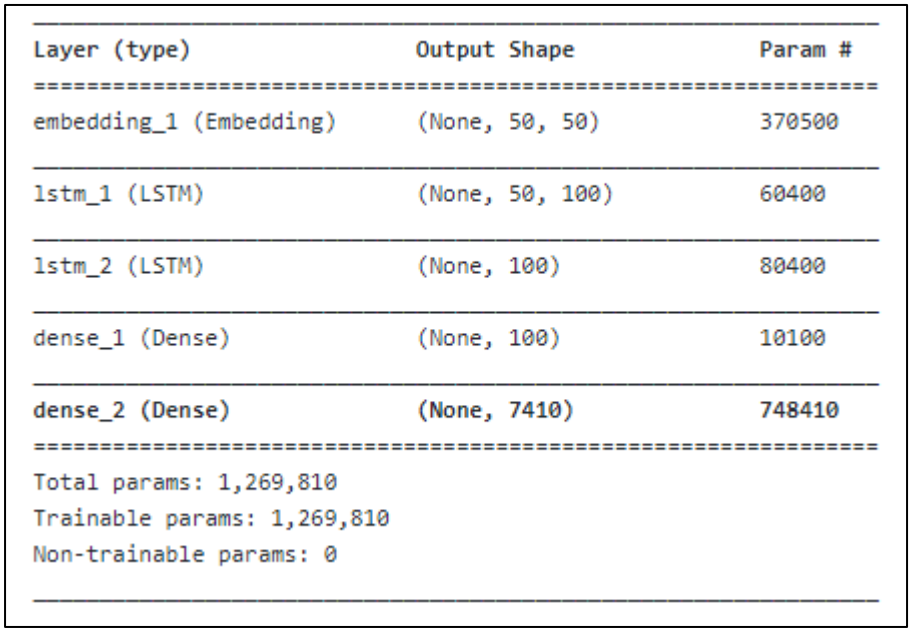

Рисунок 3.6 - Вивід результатів тренування моделі

Далі я ускладнюю модель, додаючи категоріальну втрату ентропії, необхідну для відповідності моделі. Технічно модель вивчає багато класну класифікацію, і це є відповідною функцією втрат для цього типу проблеми. Використовується ефективна реалізація Адама до міні-пакетного градієнтного спуску та оцінюється точність моделі.

Після тестових запусків навчання я вивів оптимальні параметри. Модель підходить для даних на 100 навчальних епох зі скромним розміром партії 128, щоб збільшити швидкість роботи.

Навчання може тривати кілька годин на сучасному апаратному забезпеченні без графічних процесорів. Крім того, процес тренування можна прискорити ще з більшим розміром партії та меншими навчальними епохами (рис. 3.7).

```
# compile model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
model.fit(X, y, batch_size=128, epochs=100)
```

Рисунок 3.7 - Прискорення процесу навчання

Під час тренування також бачимо підсумки результатів, включаючи втрати та точність, оцінені з навчальних даних наприкінці кожного періоду оновлення.

В результаті тренувань отримуємо різні результати, з яких можемо зробити висновок, що точність складає лише трохи більше 50% передбачення наступного слова в послідовності, що не є поганим. Не потрібно прагнути до 100% точності, адже це вже буде модель, яка запам'ятовує текст, а я прагну отримати модель, яка фіксує сутність тексту (рис. 3.8).

```
...
Epoch 96/100
118633/118633 [==============================] - 265s - loss: 2.0324 - acc: 0.5187
Epoch 97/100
118633/118633 [==============================] - 265s - loss: 2.0136 - acc: 0.5247
Epoch 98/100
118633/118633 [==============================] - 267s - loss: 1.9956 - acc: 0.5262
Epoch 99/100
118633/118633 [==============================] - 266s - loss: 1.9812 - acc: 0.5291
Epoch 100/100
118633/118633 [==============================] - 270s - loss: 1.9709 - acc: 0.5315
```

Рисунок 3.8 - Відображення процесу тренування моделі

По завершенню виконання навчена модель зберігається у файлі. Для цього я використовую API моделі Keras, щоб зберегти модель у файлі 'model.h5' у поточному робочому каталозі.

При завантаженні моделі для прогнозування, також буде потрібно відображення слів до цілих чисел. Цей функціонал знаходиться в об'єкті Tokenizer, і ми можемо заощадити час на власну реалізацію, використовуючи Pickle (рис. 3.9).

```
1   # save the model to file
2   model.save('model.h5')
3   # save the tokenizer
4   dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

Рисунок 3.9 - Зберігання моделі у файл

3.5 Програмна реалізація модуля передбачення наступного слова у реченні

3.5.1 Програмна реалізація механізму уваги

При навчанні і роботі з моделлю всі дані представлені векторами. Це призводить до ряду проблем:

- Стискається багато інформації у векторному форматі з обмеженим розміром;
- Градієнти мають довгий шлях до кінцевого результату. Навіть LSTM моделі мають феномен забування контексту при довгих векторах.

Для вирішенням цих проблем зроблено наступне:

- Вихідне речення представляється у вигляді матриці (рис. 3.10), що вирішує проблему пропускної здатності;

- Генерація цільового речення з матриці вирішує проблему градієнтного потоку.

Речення як матриці:

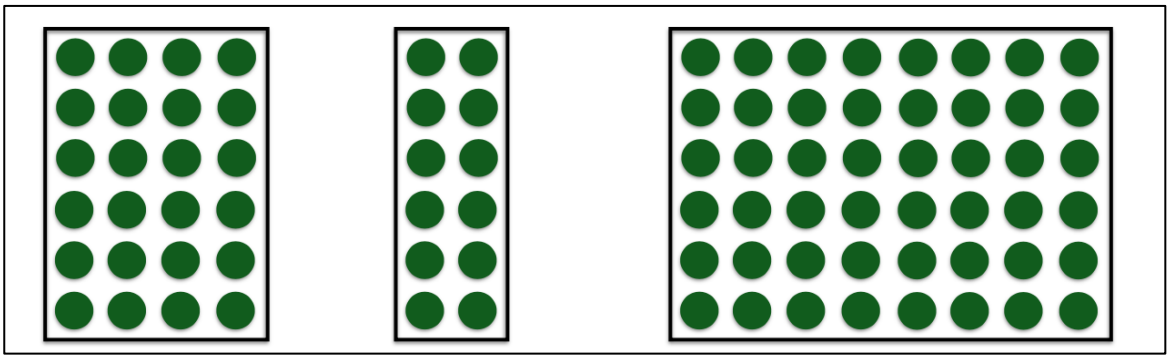

Рисунок 3.10 - Представлення вихідного речення у вигляді матриці

Представлення вихідного речення у вигляді матриці можна реалізувати з конкатенацією, де кожне окреме слово представлено n-мірним вектором (рис. 3.11). Потрібно взяти усі вектори для речення і об'єднайте їх у матрицю і отримаємо найпростішу можливу модель.

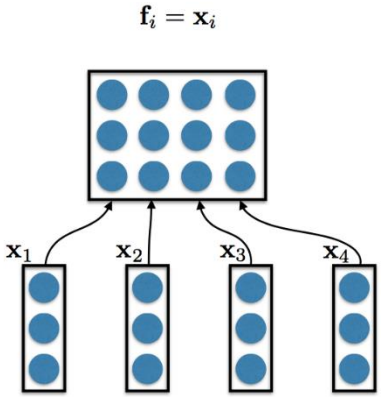

Рисунок 3.11 - Представлення слів n-мірним вектором

Альтернативний підхід використовується з конволюційними мережами, де застосовуються конволюційні мережі для перетворення наявної зчепленої

матриці, щоб отримати контекстно-залежну матрицю (рис. 3.12). Але слід зауважити, що матриці, як правило, мають операцію "згрупування" на верхньому рівні, що призводить до представлення матриці фіксованого розміру.

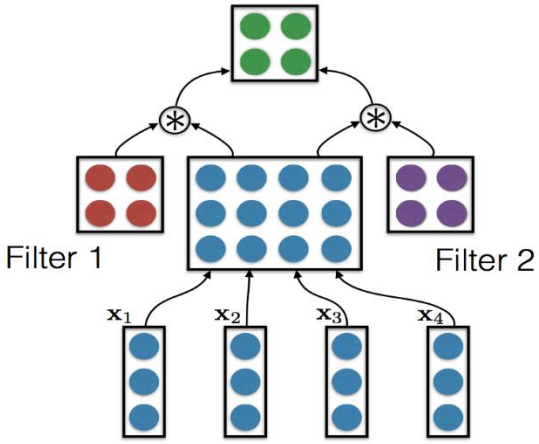

Рисунок 3.12 - Отримання контекстно-залежної матриці

При використанні двонаправленої RNN:

- Застосовано широко використовуване матричне представлення;
- Використовується одна колонка на слово;
- Кожен стовпець (слово) має дві частини, об'єднані в контекст:
    - "пряме подання", тобто слово та його лівий контекст;
    - "зворотне подання", тобто слово та його правий контекст.

Двонаправлені RNN (GRU або LSTM) використовуються, щоб читати функцію зліва направо та справа наліво для подальшого об'єднання контексту.

Для реалізації уваги потрібно генерувати вихідний вираз слово за словом за допомогою RNN мережі (рис. 3.13). На кожному кроці генерації вихідного значення RNN отримує два входи на додаток до стандартних рекурентних входів – векторне представлення попередньо сформованого вихідного

символу та вектор фіксованого розміру, який визначає формат представлення вхідної матриці. Щоб отримати вектор з фіксованим розміром з матриці, яка змінюється з часом Багданау пропонує зробити зважену суму стовпців, виходячи з того, наскільки важливі вони на поточному етапі роботи.

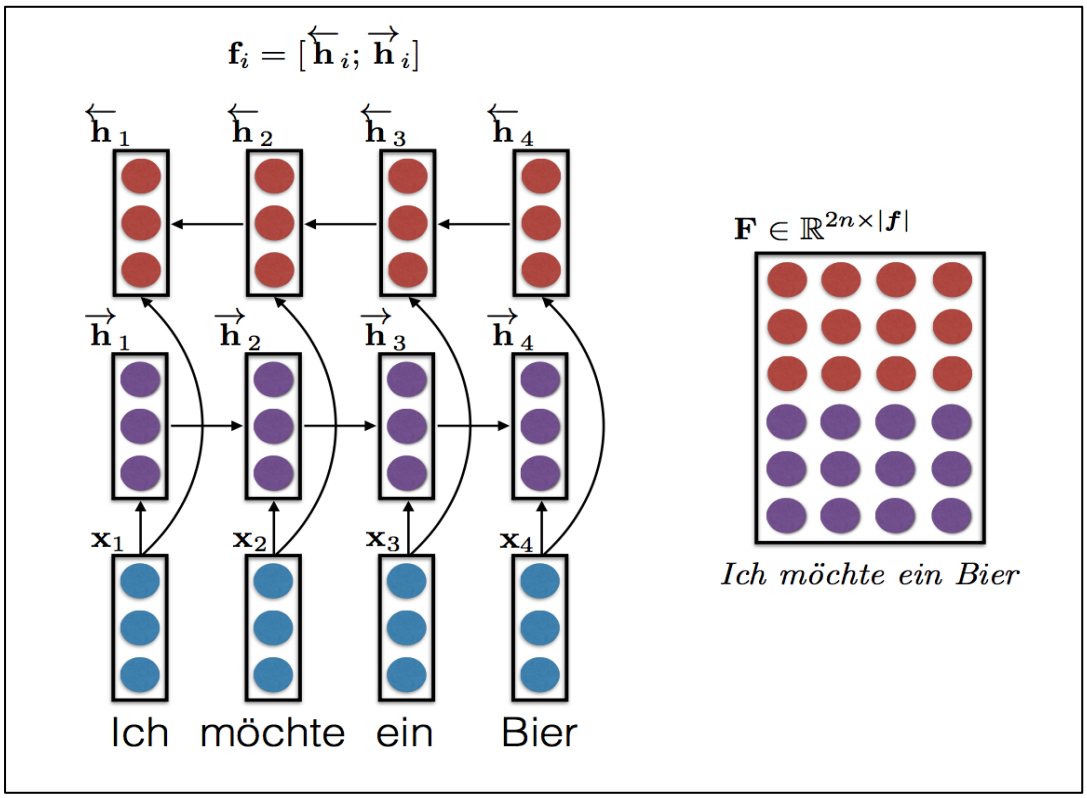

Рисунок 3.13 - Важливість вхідних стовпців на кожному кроці

3.5.2 Програмна реалізація механізму пам’яті

Для реалізації механізму пам’яті потрібно на кожному кроці генерації вихідного тексту програми мати можливість відвідувати різні слова у вхідному реченні (рис. 3.14). На кожному кроці обчислюється вага для кожного стовпця, використовуючи RNN мережу для прогнозування вихідної моделі, викликаються приховані стани. Під час обчислення наступного

вхідного значення до уваги береться точкове значення з кожного стовпця у матриці джерела для обчислення нелінійної енергії уваги. Наступним кроком отримане на попередньому кроці значення експонентується та нормалізується до одиниці для отримання вектору вхідного джерела[20] для часу t.

$\mathbf{F} = \text{EncodeAsMatrix}(\boldsymbol{f})$ (Part 1 of lecture)
$e_0 = \langle \mathbf{s} \rangle$
$\mathbf{s}_0 = \mathbf{w}$ (Learned initial state; Bahdanau uses $\mathbf{U}\overleftarrow{\mathbf{h}}_1$)
$t = 0$
while $e_t \neq \langle /\mathbf{s} \rangle$:
    $t = t + 1$
    $\mathbf{r}_t = \mathbf{V}\mathbf{s}_{t-1}$
    $\mathbf{u}_t = \mathbf{v}^\top \tanh(\mathbf{W}\mathbf{F} + \mathbf{r}_t)$ } (Compute attention; part 2 of lecture)
    $\mathbf{a}_t = \text{softmax}(\mathbf{u}_t)$
    $\mathbf{c}_t = \mathbf{F}\mathbf{a}_t$
    $\mathbf{s}_t = \text{RNN}(\mathbf{s}_{t-1}, [\mathbf{e}_{t-1}; \mathbf{c}_t])$ ($\mathbf{e}_{t-1}$ is a learned embedding of e_t)
    $\mathbf{y}_t = \text{softmax}(\mathbf{P}\mathbf{s}_t + \mathbf{b})$ ($\mathbf{P}$ and $\mathbf{b}$ are learned parameters)
    $e_t \mid \boldsymbol{e}_{<t} \sim \text{Categorical}(\mathbf{y}_t)$

Рисунок 3.14 - Загальний алгоритм обчислення уваги

Під час тренування також бачимо підсумки результатів, включаючи втрати та точність, оцінені з навчальних даних наприкінці кожного періоду оновлення.

В результаті тренувань отримуємо різні результати, з яких можемо зробити висновок, що точність складає лише трохи більше 50% передбачення наступного слова в послідовності, що не є поганим. Не потрібно прагнути до 100% точності, адже це вже буде модель, яка запам'ятовує текст, а я прагну отримати модель, яка фіксує сутність тексту.

Наступним кроком є додавання уваги до перекладу seq2seq (рис. 3.15):

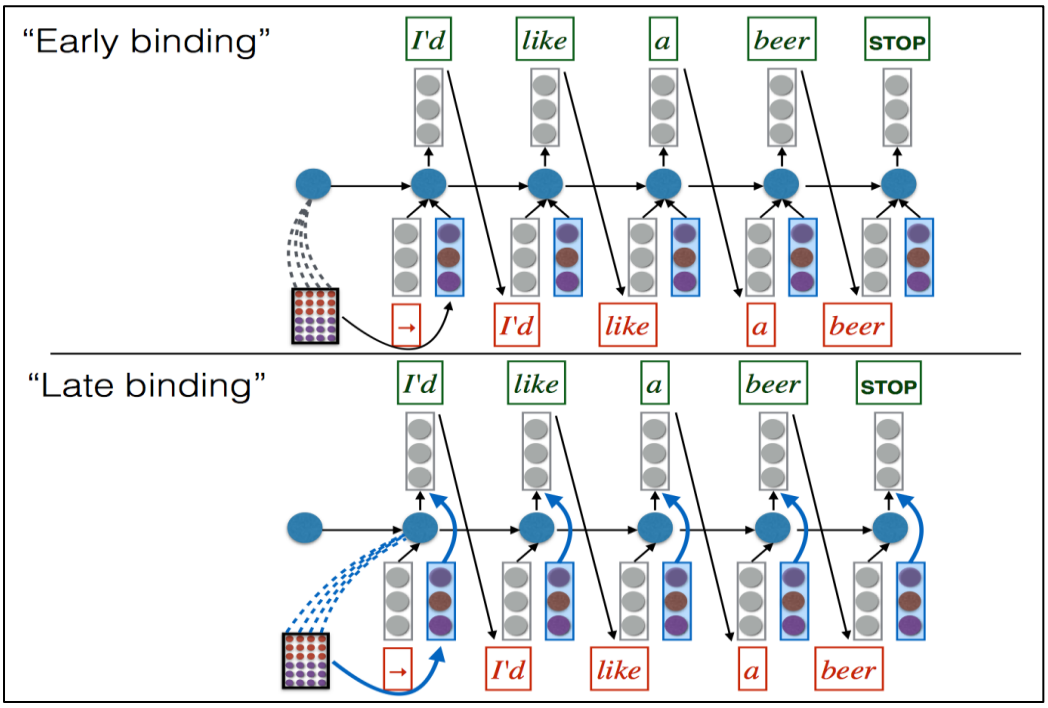

Рисунок 3.15 - Додавання уваги до перекладу seq2seq

Механізм уваги тісно пов'язаний з об'єднанням операцій в конекторах. Слід зауважити, що модель Багданау застосовує аналогічний механізм лише до "вмісту" вхідного значення. Немає очевидного упередження на користь використання діагоналей та коротких стрибків. Використання механізму уваги надає можливість отримати візуальне представлення роботи нейромережі, яку ви можете подивитися та проаналізувати додатково.

3.5.3 Отримання вхідних даних

Для підготовки вхідних даних зробимо припущення що слово вкладеного шару передбачає, що вхідні послідовності складаються з цілих чисел.

Першим кроком є присвоєння кожному слову в словнику вхідних слів унікального цілого числа з наступним кодуванням отриманої вхідної послідовності. Пізніше, при прогнозуванні результату, прогноз перетвориться на цифри і вихідне значення буде формуватися враховуючи отримане кодування на початковому етапі.

Для кодування вхідної послідовності використовувався клас Tokenizer, який наявний у фреймворку TensorFlow завдяки інтеграції з API бібліотеки Keras.

За допомогою Tokenizer треба закодувати весь навчальний набір даних, а це означає, що він знаходить всі унікальні слова у вхідній послідовності навчальних даних і присвоює кожному унікальне ціле число.

У подальшому відповідний Tokenizer використовується для кодування всіх навчальних послідовностей, перетворюючи кожну послідовність зі списку слів на список цілих чисел.

3.5.4 Обробка вхідних даних

Побудувавши словник вхідних послідовностей потрібно отримати відображення слів до цілих чисел як атрибут словника з назвою word_index в об'єкті Tokenizer.

Потрібно знати розмір отриманого словника для визначення шару вкладання пізніше при побудові структури мережі. Словниковий запас визначається, підрахувавши розмір словника зіставлення.

Словам призначаються числові індекси від одиниці до загальної кількості слів. Вбудований шар повинен виділити векторне подання для кожного слова в цьому словнику від першого індексу до найбільшого індексу і тому, що індексація масивів зрівнюється з нулем, індекс слова в кінці

лексики складе максимальне значення, що означає, що масив повинен бути довжиною на одиницю більше за максимальне значення.

Тому, для задання розміру вхідної лексики до шару вбудовування, потрібно вказувати його на одиницю більше, ніж фактичний словник.

3.5.5 Взаємодія структур для отримання та обробки передбачень

Після отримання вхідних закодованих послідовностей, потрібно розділити їх на вхідні і вихідні елементи. Це можна зробити за допомогою нарізки масиву.

Після відокремлення нам потрібно одне гаряче кодування вихідного слова. Це означає перетворення його від цілого до вектора з нульовими значеннями, по одному для кожного слова в словнику та з одиничним значенням для позначення конкретного еталонного слова в масиві значень.

Це означає, що модель навчається прогнозувати розподіл ймовірностей для наступного слова, а також визначає істину, від якої слід вчитися. Нульове значення для всіх слів, за винятком фактичного слова, що наводиться далі як правильне еталонне значення.

Бібліотека Keras забезпечує метод to_categorical, який може використовуватися для гарячого кодування вихідних слів для кожної пари послідовності вводу та виводу.

На останньому кроці потрібно визначити шар вбудовування та скільки пам’яті будуть займати вхідні послідовності. Знаючи наперед, скільки всього існує у словнику слів, не потрібно розробляти модель, але загальним способом вказати це є використання другого розміру (кількість стовпчиків) форми вхідних даних. Таким чином, якщо змінюється довжина

послідовностей під час навчання моделі на вхідних даних, вам не потрібно змінювати код завантаження даних.

3.6 Тестування розроблених модулів

Після отримання навченої мовної моделі можна її використовувати для створення нових послідовностей тексту, які мають ті ж статистичні властивості, що і вхідний текст.

Це непрактично, принаймні, не для даного випадку, але він дає конкретний приклад того, чому навчилась отримана мовна модель.

Для початку потрібно завантажити навчальні послідовності.

Для тестування будемо використовувати той же код з попереднього розділу, щоб завантажити послідовності даних навчального тексту, зокрема, функцію load_doc (рис. 3.16)

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text
# load cleaned text sequences
in_filename = 'republic_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
```

Рисунок 3.16 - Завантаження мовних послідовностей

Для подальшого тестування потрібно підібрати вхідний текст, щоб мати можливість вибрати послідовність джерел як вхід до моделі для створення нової послідовності тексту. Для введення моделі було підібрано сто слів.

Пізніше потрібно буде вказати очікувану довжину введення. Це можна зробити, визначивши це з вхідних послідовностей шляхом обчислення довжини одного рядка завантажених даних і віднімання одиниці для очікуваного вихідного слова, яке також знаходиться на одній лінії. Після цього потрібно завантажити модель з файлу.

Бібліотека Keras має функцію load_model для завантаження моделі, готової до використання. Крім того потрібно завантажити токенізатор з файлу за допомогою API бібліотеки Pickle, після чого завантажена модель готова до використання.

Першим кроком у створенні тексту є підготовка даних для введення. Для цього я вибрав випадкову частину вхідного тексту. Після цього надрукував його так, щоб мати певне уявлення про те, що було використано. Далі будемо генерувати нові слова, по одному на кожному кроці. По-перше, текст насіння повинен бути закодований цілими числами, використовуючи той же токенізатор, який використовувався під час тренування моделі. Модель може передбачити наступне слово безпосередньо, викликавши метод predict_classes, який поверне індекс слова з найбільшою ймовірністю. Так зможемо знайти індекс у словниках класу tokenizer, щоб отримати пов'язане слово (рис. 3.17).

```
# generate a sequence from a language model
def generate_seq(model, tokenizer, seq_length, seed_text, n_words):
    result = list()
    in_text = seed_text
    # generate a fixed number of words
    for _ in range(n_words):
        # encode the text as integer
        encoded = tokenizer.texts_to_sequences([in_text])[0]
        # truncate sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
        # predict probabilities for each word
        yhat = model.predict_classes(encoded, verbose=0)
        # map predicted word index to word
        out_word = ''
        for word, index in tokenizer.word_index.items():
            if index == yhat:
                out_word = word
                break
        # append to input
        in_text += ' ' + out_word
        result.append(out_word)
    return ' '.join(result)
```

Рисунок 3.17 - Генерація послідовностей з мовної моделі

Після цього ми можемо додати це слово до насіннєвого тексту та повторити процес.

Важливо, що вхідна послідовність буде надто довгою. Потрібно скоротити її до потрібної довжини після того, як вхідна послідовність буде закодована до цілих чисел. Бібліотека Keras пропонує функцію pad_sequences, яку потрібно використати для виконання цього скорочення.

Результат потрібно загорнути в функцію gener_seq, яка приймає вхідну модель, токенізатор, довжину послідовності вводу, текст насіння та кількість слів, які потрібно генерувати. Потім вона повертає послідовність слів, згенерованих моделлю.

Тепер потрібно сформувати послідовність нових слів, враховуючи деякі насіннєві тексти. Запустивши приклад, спочатку друкується текст насіння. Можна побачити, що текст виглядає розумним. Справді, додавання конкатенації допоможе інтерпретувати насіння та згенерований текст. Тим не менше, сформований текст отримує потрібні слова в правильному порядку.

4. АПРОБАЦІЯ МЕТОДУ ПРИСКОРЕННЯ СТАТИСТИЧНОГО МОДЕЛЮВАННЯ МОВ НА ОСНОВІ ВИКОРИСТАННЯ ФРЕЙМВОРКУ TENSORFLOW

4.1. Апробація моделі рекурентної мережі

Уявіть собі такий сценарій в програмної інженерії: існує великий репозиторій програм з величезною кількістю високоякісного вихідного коду, який добре відкоментований і документований. Дуже потужна машина (наприклад, глибока нейронна мережа) навчається відображати опис проблем на природній мові в вихідний код. У процесі розробки, користувач висловлює свої наміри натуральною мовою (схоже на щось в репозиторії). Після чого навчена машина автоматично виводить необхідний код в якості рішення.

Більш захоплюючою властивістю описаного вище процесу є те, що він виходить цілісним, що вимагає трохи (якщо взагалі вимагає) знання програмування від людини і є абсолютно незалежним від мови програмування. Єдине, що необхідно - це подати на вхід програми запит у вигляді символів. Навчена мережа автоматично читає запит на природній мові символ за символом, щоб зрозуміти наміри користувача і потім генерує код в аналогічній манері. Оскільки навчена мережа не є простим пошуком за кодом, то згенерований код відрізняється від уже наявних зразків, що привносить в процес більше гнучкості, хоча і уразливості теж. Код стає готовим до використання з невеликим пост-редагуванням.

Даний сценарій автоматичної генерації програм вже давно є мрією програмної інженерії (SE) і близько пов'язаний з великою кількістю завдань SE, таких як пошук алгоритму, допомога в програмуванні. Однак, традиційні підходи зазвичай слабкі в області автоматизації і абстракції. Наприклад,

Manna et al. пропонує дедуктивні підходи, Flener et al. - індуктивні; ці методи вимагають створених людьми специфікацій. Генерація програм за допомогою генетичного програмування може автоматично проводити пошук в просторі можливих програм-кандидатів (неефективно), але вона також вимагає надання ретельно вибраних операцій мутації і кросовера. Програмування на природній мові, що розвивається в останнє десятиліття, швидше схоже на «псевдокомпіляцію», де природна мова є низкорівневою абстракцією.

В наш час, програмні артефакти, включаючи код і документацію, стали «Big data» (наприклад, Github, Sourceforge). Надаючи досить навчальних даних у вигляді коду з відповідними коментарями та документами, вони роблять принципово можливим тренування моделі, що породжує програми на основі природної мови. У той же час, спільнота, яке спеціалізується обробкою природної мови (NLP), спостерігає значні прориви і вражаючі успіхи в великій кількості завдань, включаючи відповіді на питання, машинний переклад або ж генерація опису зображень. Цей міждисциплінарний прогрес приніс нові можливості в область автоматичної генерації програм.

У даній дипломній роботі, я розглянув варіант навчання мережі, здійсненність генерації виконуваного, функціонально пов'язаного коду за допомогою RNN. Емпіричний аналіз розкриває механізм, як RNN можуть досягти мети. Також я уявив кілька сценаріїв, де така техніка може допомогти у вирішенні реальних завдань програмної інженерії, і розглянув довготривалі дослідницькі завдання. Хоча, я визнаю те, що до використовування на практиці методів наскрізної генерації програм ще чекає тривалий шлях, я вірю, що це стане реальністю в найближчі десятиліття.

Серед усього розмаїття методів машинного навчання, глибокі нейронні мережі (також відомі як глибоке навчання) є одним з новітніх революційних досягнень, які характеризуються своєю здатністю автоматично навчатися дуже складним ознакам.

Для наскрізної генерації програми, я вважаю, що краще використовувати рекурентні нейронні мережі (RNN), які зручні для моделювання тимчасових серій даних (наприклад, послідовності символів) за своєю ітеративною природою. RNN зазвичай мають один або більше прихованих шарів, дискретно змінюються на кожному часовому кроці згідно вхідних даних (рис. 4.1). Адаптація приклада. (A) Вхідна послідовність, (b) Вихідна послідовність.

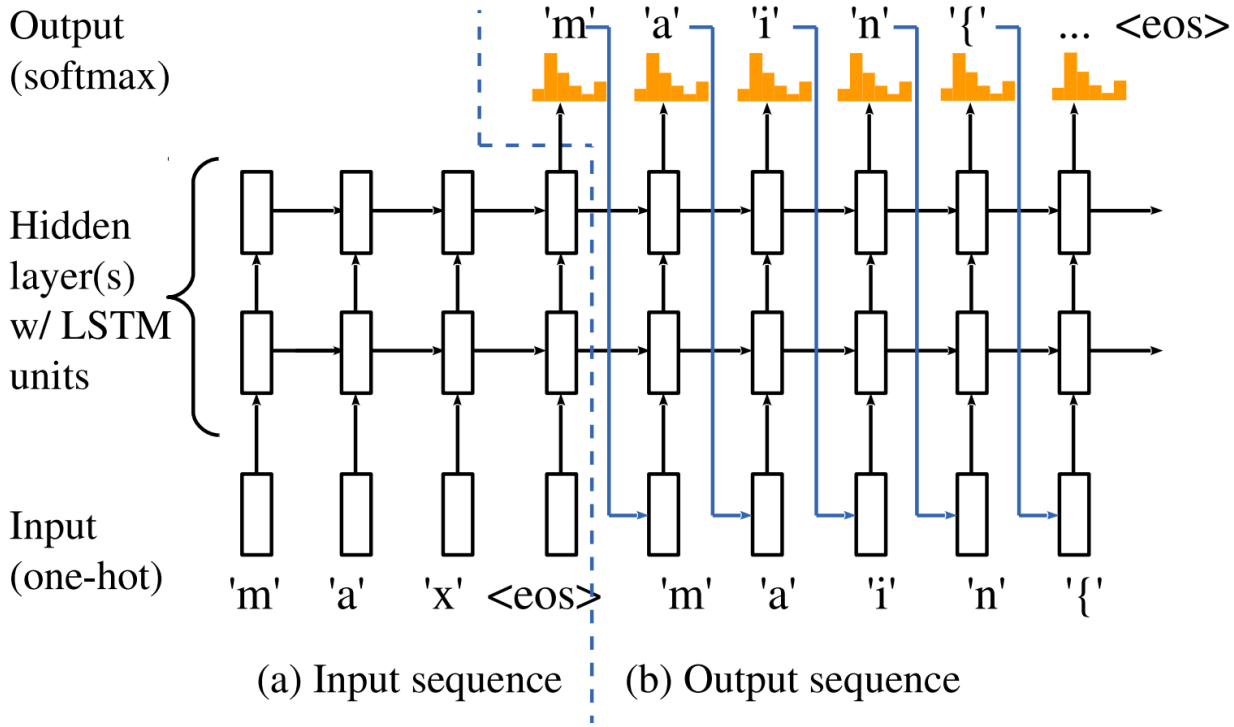

Рисунок 4.1 - Рекурентна нейронна мережа, що відображає послідовність в послідовність

Теоретичний аналіз показує, що рекурентна нейронна мережа еквівалентна машині Тьюринга. Звичайно, тренування RNN в попередні роки

було складною задачею у зв'язку з проблемою зникаючого градієнта. Модулі довгої короткочасної пам'яті (long short-term memory, LSTM) або ж імпульсні модулі (gated units) створені для додавання балансу між збереженням попереднього стану і запам'ятовуванням нової інформації на поточному часовому кроці, дозволяють навчати RNN набагато легше.

На цьому заснований принцип RNN моделі для перетворення послідовності в послідовність. Ідея полягає в первісному читанні вхідної послідовності, що закінчується спеціальним символом <eos> (кінець послідовності). Для виведення, RNN застосовує softmax-шар для кожного тимчасового кроку, який передбачає ймовірність, з якою кожен символ може зустрітися на поточному кроці. Далі вибирається символ з найбільшою ймовірністю і подається мережі на вхід для наступного часового кроку. Даний процес триває ітеративно, поки мережею не буде згенеровано спеціальний символ <eos>.

Така архітектура RNN може бути застосована до послідовностей з різною гранулярністю, наприклад, рівня слів, частин слів і т.д. Зокрема, отримана RNN модель рівня символів несподівано виявляється видатною і трохи вражає своєю продуктивністю. Успішні програми генерують тексти, музику і навіть C-код для Linux-програм. Емпіричні дослідження показали, що RNN особливо гарні в моделюванні синтаксичних аспектів, таких як сполучення дужок, відступів і т.п. Це працює часто як автомати з магазинною пам'яттю, але, схоже, вони менш здатні охопити семантику - згенерований Linux-код, наприклад, виглядає правдоподібно, але не може бути скомпільований і не узгоджений функціонально.

Таким чином, мені цікаво, чи може RNN генерувати виконуваний, функціонально пов'язаний код, який є сутністю завдання створення реального програмного забезпечення.

Для досягнення цієї мети, я використовував набір даних з онлайн системи навчання програмуванню (online judge system, OJ), націленої на навчання студентів азам програмування. OJ система пропонує вирішити різні проблеми. Студенти відсилають туди свій вихідний код, вирішуючи певну проблему, а OJ валідує його автоматично (виконуючи). Я звернув увагу, що програми, які вирішують конкретне завдання, мають однакову функціональність, що робить цей набір даних особливо підходящим для навчання нейронної мережі, яка генерує програми.

4.2 Апробація модуля тренування мовної моделі

Я навчав нейронну мережу чотирьом різним завданням, кожне з яких містило понад 500 різних зразків вихідного коду. Після попередньої обробки, до програми додавався короткий коментар, наприклад «знайти максимальне і наступне за ним по величині число», який служив вхідною послідовністю (рис. 4.2). Програма, яка вирішує певну проблему, служила в якості вихідної послідовності (рис. 4.3).

На рисунку 4.3 наведено приклад коду, згенерований RNN. Після швидкого аналізу я знайшов цей код в основному придатним до виконання: після невеликого пост-коригування чотирьох символів з приблизно 280, програма стала правильною і функціонально коректною.

Перевіривши дані для навчання, я знайшов, що в них відсутній точно такий же код в тренувальному наборі, що виключає можливість того, що RNN відпрацювала за допомогою точного співпадіння. Далі, я використав ccfinder, щоб знайти найбільш схожий код в тренувальному наборі. Два з них показані на рисунку 4.4, і ці результати є особливо цікавими. Далі я пропоную моє пояснення деяких аспектів програми.

Про структуру вихідної програми. На рисунку 4.5 показаний найбільш схожий за структурою код. Згенерований фрагмент реалізує той же алгоритм - сканування масиву двічі, щоб знайти максимальний і наступне за ним за значенням число, відповідно. Зверну увагу, що дві структури (абстрактні синтаксичні дерева) не відповідають точно, оскільки є відмінності у визначеннях змінних. Більш цікавою деталлю є, що RNN визначила, що "i <n" і "i <= n-1" синонімічні в циклі for і в цьому згенерований код не відповідає коду прикладу (b) з тренувального набору, однак залишається коректним.

Про ідентифікатори змінних. Тренувальний приклад з найбільш схожим набором ідентифікаторів показаний на рисунку 4.5. Згенерований код використовує такий же ідентифікатор "a" для масиву, max1 і max2 для

```
#include<stdio.h>
void main()
{       (1) n
    int x,a[100],i,max1,max2;
    scanf("%d",&n);
    for(i=0;i<=n-1;i++)
    {
        scanf("%d",&a[i]);
        if (a[i]>max1)
            max2=a[i];}
        for(i=1;i<=n;i++)
    {               (2) 2   (3) <
        if(a[i]>max&&a[i]==max1)
            max2=a[i];
    }

    printf("%d\n%d",max1,max2);
    return 0; (4) [del]
}
```

Рисунок 4.2 Код, згенерований RNN. Код майже коректний, за винятком 4-х неправильних символів (при загальній кількості приблизно 280). Ці місця виділені на малюнку

кешування двох необхідних чисел, однак далі, структура розходиться. Проте, отримана мережа усвідомлює, які ідентифікатори змінних були згенеровані і залишається узгодженою до самого кінця програми.

Про стиль програми. Я не знайшов в тренувальному наборі коду з таким же стилем відступів, перенесення рядків і т.п. Це має сенс, оскільки тренувальні програми написані початківцями програмістами, які не дотримуються стандартних угод про стилі, відповідно, мережа не має уявлення про те, який стиль «правильний». Однак, оскільки всі тренувальні приклади є «коректними» програмами, наша мережа відчувала невеликі складнощі в навчанні синтаксису С програм, і згенерований код майже може бути скомпільований.

```
#include<stdio.h>
int main(){
     int n,i,j,sz[100],max=0,ci=0;
     scanf("%d",&n);
    for(i=0;i<n;i++){
        scanf("%d",&sz[i]);
        if(sz[i]>max){
           max=sz[i];}}
      for(i=0;i<n;i++){
 if(sz[i]>ci&&sz[i]<max){
        ci=sz[i];}}
 printf("%d\n%d",max,ci);
return 0;
}
```

Рисунок 4.3 Код з найбільш схожою структурою з тренувального набору. Знайдений за допомогою ccfinder

Зверніть увагу, що ми зберегли всі відступи, прогалини і переноси рядків. Чотири знайдені помилки: (1) Ідентифікатор "x" повинен бути

замінений на "n". (2) "max" має бути замінено на "max2". (3) "==" замінюється на "<". (4) Функція повинна повертати тип void.

Відповідно до наведеного вище аналізу, був отриманий базовий принцип, згідно з яким RNN в змозі генерувати програми. Спочатку RNN розпізнає короткий коментар «знайти максимальне і наступне за ним по величині число», яке передує коду на вході. Я хотів би відзначити, що в даній

```
#include<stdio.h>
void main()
{
    int n,i,a[100],j,max1,max2;
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    max1=a[0];
    for(i=0;i<n;i++)
    {
        if(a[i]>max1)
            max1=a[i];
    }
    for(i=0;i<n;i++)
    {
        if(max1==a[i])
            j=i;
    }
    if(max1!=a[0])
    max2=a[0];
    else max2=a[1];
    for(i=0;i<n;i++)
    {
        if(i==j) continue;
        if(a[i]>max2)
            max2=a[i];
    }
        printf("%d\n%d",max1,max2);
}
```

(c)
Training
sample 2

Рисунок 4.4 - Код з найбільш схожими ідентифікаторами, також знайдений за допомогою утиліти ccfinder

роботі, RNN не розуміє сенсу цієї пропозиції, але через читання цього короткого коментаря, RNN перемикає свої приховані стану на генерацію коду з тієї функціональністю, яка потрібна. Для кожної функції, RNN усвідомлює різні аспекти можливої програми, включаючи структуру програми, ідентифікатори і т.п. В процесі генерації, мережа вибирає найбільш ймовірний символ, відповідно до попередніх символів, також відповідно до введення.

Зокрема, RNN має можливість змішувати різні структури і ідентифікатори, які, тим не менш, залишаються узгодженими між собою.

Поки просте і попереднє, моє дослідження і аналіз показують блискучу картинку того, як може відбуватися наскрізна генерація програм з використанням глибоких нейронних мереж. Відзначу кілька сценаріїв, в яких глибоке навчання може принести користь реальній програмній інженерії, що породжує напрямки для подальших досліджень.

Що до розуміння мінливих намірів користувача, то дане дослідження добре показує, що RNN мережа здатна розпізнати відповідний намір користувача і згенерувати відповідний код. Однак, в практиці програмування, я часто стикаюся з мінливими вимогами користувачів. Для вирішення цієї проблеми, прямолінійним рішенням є тренування параметричного генератора коду з аргументами (імена файлів, протоколи), які неявно потребують використання природної мови. Для вирішення більш складних випадків, я міг би навчити мережу спочатку створювати примітивні фрагменти коду, а потім склеювати їх разом. Наприклад, якщо мережа навчена генерувати код для знаходження максимального числа, для знаходження мінімального числа, то вона повинна бути здатна згенерувати ці два фрагменти послідовно, якщо прочитає інструкцію «знайти максимальне і мінімальне значення».

Об'єднувати декілька джерел намірів користувача. В процесі розробки програмного забезпечення, розробники зазвичай знаходять свої ділянки коду в залежності від контексту (наприклад, попередньо визначені змінні, послідовності виклику API), доповнюючи функціональність в міру необхідності. У таких випадках можна було б навчити мережу заповнювати відсутні блоки коду. Наприклад, стандартний спосіб читання текстового файлу в Java включає в себе створення FileReader, BefferedReader, читання рядків у файлі, закриття файлу і обробку винятків. Такі стандартні конвеєри можуть бути згенеровано автоматично нейронними мережами, на основі аналізу контекстного коду.

4.3 Апробація модуля передбачення наступного слова у реченні

Незважаючи на багатообіцяюче майбутнє RNN мереж в області генерації вихідного коду, зусилля повинні бути прикладені в декількох напрямках, включаючи програмну інженерію, NLP і машинне навчання. Найбільш важливими питаннями в співтоваристві програмній інженерії є визначення намірів користувача і підготовка даних для навчання. Потрібно розуміти як визначити функціональність того, що потрібно згенерувати чи як можна специфікувати аргументи функції. Потрібно вдосконалити процес збору набору даних, який буде не тільки досить великим але і досить інформативним для навчання, але також є досить чистим і не містить занадто багато шуму. З іншого боку, спільноти NLP і машинного навчання постійно вдосконалюють архітектури нейронних мереж. Мережі засновані на увазі, наприклад, нещодавно були запропоновані для того, щоб пом'якшити проблему довгих вхідних послідовностей, які не можуть бути скомпоновані в вектор з фіксованою довжиною. Також додаткові дослідження необхідні

для розуміння обсягу пам'яті RNN мережі, генерації даних з більш узгодженою семантикою, або навіть ревізії вже згенерованих даних і т.д.

Використання RNN мережі для генерації програм значно відрізняється від їх написання людьми. На даний момент виглядає нереальним навчити будь-яку мережу, включаючи глибокі нейронні мережі, щоб вона була спроможною повністю розуміти природні мови або мови програмування. Однак підтримувана існуючими доказами в літературі і конкретним випадком, розглянутим у цій роботі, я вважаю, що наскрізна генерація програм буде можлива в майбутньому.

В останні роки ми стали свідками народження аналізу програм, заснованого на глибоких нейронних мережах. Моя попередня робота вивчала векторне подання програм, що служило підготовчою фазою для глибокого навчання. Крім того я пропоную засновані на деревах згорткові нейронні мережі для класифікації програм по функціональності і визначення певних патернів в вихідному коді. Zaremba et al. використовував RNN для оцінки виведення обмежених програм на Python. Allamanis et al. використовував векторне подання для того, щоб пропонувати імена методів. Всі вищевказані моделі є дискримінантними, що можуть розглядатися в якості завдання класифікації. Karpathy et al. тренував RNN-базовану модель на C-коді, що максимізує об'єднану ймовірність програми. На відміну від вищезгаданих досліджень, дана робота досліджує, чи можуть нейронні моделі синтезувати виконувані, функціонально узгоджені програми, що вимагає більшої відповідності намірам користувачів і більш правильної внутрішньої структури коду.

У даній роботі я навчив рекурентну нейронну мережу (RNN) генерувати майже виконуваний, функціонально узгоджений вихідний код. Нашим первинним завданням було продемонструвати можливість

автоматичної наскрізний генерації програм. Проаналізувавши механізм роботи RNN, я запропонував кілька сценаріїв, де такі техніки можуть бути корисні для завдань розробки програмного забезпечення в найближчі десятиліття. Я закликаю до подальших міждисциплінарних досліджень в цьому новому напрямку.

4.4 Оцінка результатів використання запропонованого методу

Цікава частина полягає, звичайно, в тому, наскільки швидко TPU дійсно є. TensorFlow має сховище моделей GitHub для TPU, які, як відомо, добре працюють. Нижче ми повідомляємо про експерименти з ResNet та Inception. Ми також хотіли бачити, як функціонує модель, яка ще не оптимізована для TPU, тому ми адаптували модель класифікації тексту з використанням LSTM для роботи на TPU. Загалом, Google рекомендує використовувати більші моделі (дивіться, коли використовувати TPU). Це менша модель, тому особливо цікаво було б з'ясувати, чи зможуть TPU все ще забезпечити вигоду.

Для всіх моделей ми порівнювали швидкість навчання на одному Cloud TPU на одному графічному процесорі Nvidia P100 і V100. Ми відзначаємо, що ретельне порівняння повинно також включати кінцеву якість та конвергенцію моделі, крім простої пропускної спроможності. Наші експерименти маються на увазі як перший огляд, і ми залишмо докладний аналіз майбутньої роботи.

Експерименти для TPU та P100 були запущені на платформі Google Cloud Platform на 16-ти примірниках (16 vCPUs Intel Haswell, 60 Гб пам'яті). Для V100 GPU ми використовували P3.2xlarge (8 vCPUs, 60 Гб пам'яті) екземпляри на AWS. У всіх системах працював Ubuntu 16.04. Для TPU, ми

встановили TensorFlow 1.6.0-rc1 з сховища PyPi. Експерименти з GPU виконувалися за допомогою nvidia-docker, використовуючи зображення TensorFlow 1.5 (tensorflow: 1.5.0-gpu-py3), які включають CUDA 9.0 та cuDNN 7.0.

Давайте спочатку подивимося на продуктивність моделей, які офіційно оптимізовані для TPU. Нижче ви можете побачити продуктивність з точки зору зображення в секунду.

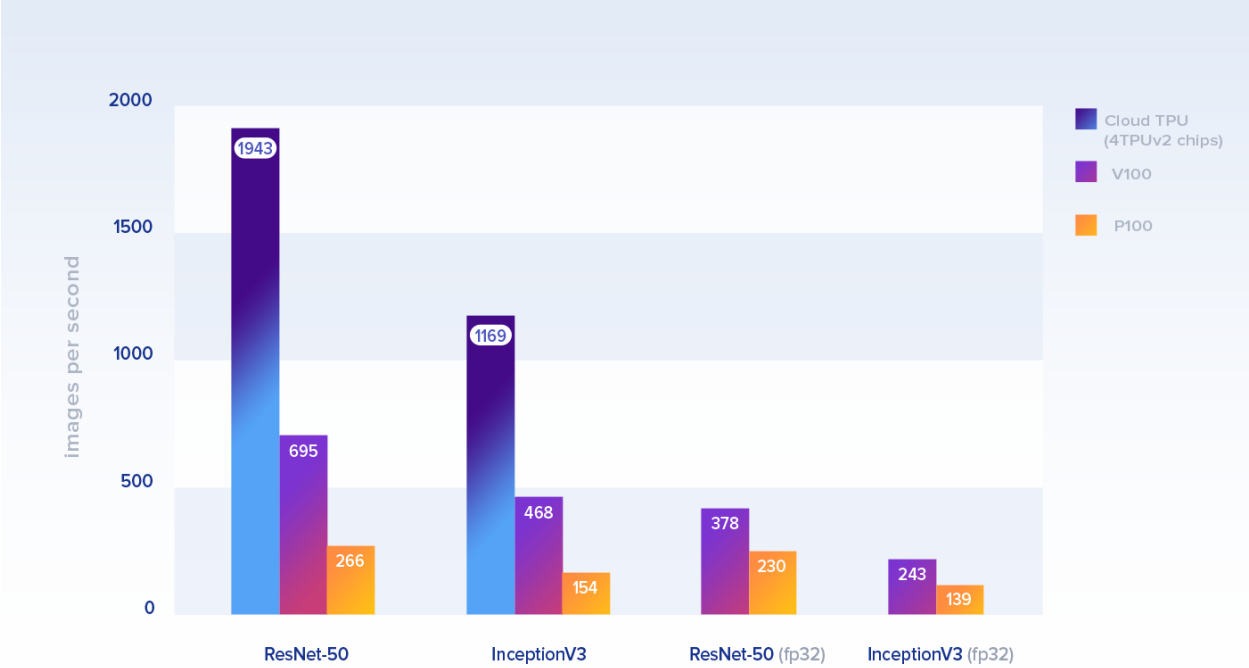

Рисунок 4.5 - Розміри пакетів становили 1024 для TPU та 128 для GPU. Для GPU я використовував реалізацію з репозиторію тестів TensorFlow за допомогою прапора 'use_fp16 = true' для бітів, не позначених 'fp32'. Отже, дві групи барів ліворуч порівнюють тренування з підвищеною точністю. Дані щодо тренувань були підробленим набором даних Imagenet, що надається компанією Google, що зберігається у хмарному сховищі (для TPU) та на локальних дисках (для графічних процесорів)

Далі я ускладнюю модель, додаючи категоріальну втрату ентропії, необхідну для відповідності моделі. Технічно модель вивчає багатокласну класифікацію, і це є відповідною функцією втрат для цього типу проблеми. Використовується ефективна реалізація Адама до міні-пакетного градієнтного спуску та оцінюється точність моделі.

Після тестових запусків навчання я вивів оптимальні параметри. Модель підходить для даних на 100 навчальних епох зі скромним розміром партії 128, щоб збільшити швидкість роботи.

Навчання може тривати кілька годин на сучасному апаратному забезпеченні без графічних процесорів. Крім того, процес тренування можна прискорити ще з більшим розміром партії та меншими навчальними епохами.

Під час тренування також бачимо підсумки результатів, включаючи втрати та точність, оцінені з навчальних даних наприкінці кожного періоду оновлення.

В результаті тренувань отримуємо різні результати, з яких можемо зробити висновок, що точність складає лише трохи більше 50% передбачення наступного слова в послідовності, що не є поганим. Не потрібно прагнути до 100% точності, адже це вже буде модель, яка запам'ятовує текст, а я прагну отримати модель, яка фіксує сутність тексту.

По завершенню виконання навчена модель зберігається у файлі. Для цього я використовую API моделі Keras, щоб зберегти модель у файлі 'model.h5' у поточному робочому каталозі.

При завантаженні моделі для прогнозування, також буде потрібно відображення слів до цілих чисел. Цей функціонал знаходиться в об'єкті Tokenizer, і ми можемо заощадити час на власну реалізацію, використовуючи Pickle.

При навчанні і роботі з моделлю всі дані представлені векторами. Це призводить до ряду проблем:

- Стискається багато інформації у векторному форматі з обмеженим розміром;
- Градієнти мають довгий шлях до кінцевого результату. Навіть ЛСТМ моделі мають феномен забування контексту при довгих векторах.

Для вирішенням цих проблем зроблено наступне:

- Вихідне речення представляється у вигляді матриці (рис. 3.10), що вирішує проблему пропускної здатності;
- Генерація цільового речення з матриці вирішує проблему градієнтного потоку.

Представлення вихідного речення у вигляді матриці можна реалізувати з конкатенацією, де кожне окреме слово представлено n-мірним вектором. Потрібно взяти усі вектори для речення і об'єднайте їх у матрицю і отримаємо найпростішу можливу модель.

Альтернативний підхід використовується з конволюційними мережами, де застосовуються конволюційні мережі для перетворення наявної зчепленої матриці, щоб отримати контекстно-залежну матрицю. Але слід зауважити, що конвенети, як правило, мають операцію "згрупування" на верхньому рівні, що призводить до представлення матриці фіксованого розміру.

При використанні двонаправленої RNN:

- Застосовано широко використовуване матричне представлення;
- Використовується одна колонка на слово;
- Кожен стовпець (слово) має дві частини, об'єднані в контекст:
    - "пряме подання", тобто слово та його лівий контекст;
    - "зворотне подання", тобто слово та його правий контекст.

Двонаправлені RNN (GRU або LSTM) використовуються, щоб читати функцію зліва направо та справа наліво для подальшого об'єднання контексту.

Для реалізації уваги потрібно генерувати вихідний вираз слово за словом за допомогою RNN мережі. На кожному кроці генерації вихідного значення RNN отримує два входи на додаток до стандартних рекурентних входів – векторне представлення попередньо сформованого вихідного символу та вектор фіксованого розміру, який визначає формат представлення вхідної матриці. Щоб отримати вектор з фіксованим розміром з матриці, яка змінюється з часом Багданау пропонує зробити зважену суму стовпців, виходячи з того, наскільки важливі вони на поточному етапі роботи.

Для реалізації механізму пам’яті потрібно на кожному кроці генерації вихідного тексту програми мати можливість відвідувати різні слова у вхідному реченні. На кожному кроці обчислюється вага для кожного стовпця, використовуючи RNN мережу для прогнозування вихідної моделі, викликаються приховані стани. Під час обчислення наступного вхідного значення до уваги береться точкове значення з кожного стовпця у матриці джерела для обчислення нелінійної енергії уваги. Наступним кроком отримане на попередньому кроці значення експонентується та нормалізується до одиниці для отримання вектору вхідного джерела для часу t.

Механізм уваги тісно пов'язаний з об'єднанням операцій в коннекторах. Слід зауважити, що модель Багданау застусовує аналогічний механізм лише до "вмісту" вхідного значення. Немає очевидного упередження на користь використання діагоналей та коротких стрибків. Використання механізму уваги надає можливість отримати візуальне представлення роботи нейромережі, яку ви можете подивитися та проаналізувати додатково.

У ResNet-50 єдиний Cloud TPU (що містить 4 мікросхеми TPUv2 і 64 ГБ оперативної пам'яті) є ~ 7,3 швидше, ніж один P100 і ~ 2,8 рази швидше, ніж V100. Для InceptionV3 прискорення майже однакове (~ 7,6 та ~ 2,5, відповідно). З високою точністю (fp32) V100 втрачає велику швидкість. Зауважте, що тренування моделі з цією точністю на TPU взагалі неможливе, оскільки він підтримує лише змішану точність обчислень.

ResNet-50 Performance				
	TPUv2	**V100 fp16**	**P100**	**V100**
Cloud	Google	AWS	Google	AWS
Price *$ per hour*	7.26	3.06	1.58	3.06
images / second	1943	695	230	378
Performance *images/s per $*	268	227	146	124

Рисунок 4.6 - Зрозуміло, що за рамки простої швидкості необхідно враховувати ціну. Таблиця показує продуктивність, нормовану для ціноутворення на вимогу з розрахунком за секунду. ТПУ все ще випереджає

Інший варіант інтерпретування механізму уваги полягає в тому що він просто дає мережевий доступ до своєї внутрішньої пам'яті, яка є прихованим станом кодувача. У цій інтерпретації, замість того, щоб вибрати, до чого "брати участь", мережа вибирає те, що потрібно витягнути з пам'яті. На відміну від типової пам'яті, механізм доступу до пам'яті тут є м'яким, що означає, що мережа отримує зважену комбінацію всіх розташувань пам'яті, а

не значення з одного дискретного розташування. М’яке отримання доступу до пам'яті має вигоду в тому, що ми можемо легко тренувати мережу повноцінним способом, використовуючи зворотне поширення (хоча існують інші підходи, за яких градієнти обчислюються методами вибірки замість зворотного розповсюдження).

Самі механізми пам'яті мають набагато довшу історію. Прихований стан стандартної рекурентної нейронної мережі сам по собі є типом внутрішньої пам'яті. RNN страждають від втрати градієнта, що перешкоджає їм вивчати довготривалі залежності. На фоні цьому LSTM виглядає краще, використовуючи механізм стримування, який дозволяє видаляти і оновлювати явні області пам'яті.

Тенденція до більш складних структур пам'яті зараз триває. "End-To-End" мережі дозволяють мережі кілька разів прочитати ту ж вхідну послідовність, перш ніж робити генерацію, оновлюючи вміст пам'яті на кожному кроці. Наприклад, відповідаючи на запитання, зробивши декілька розумних кроків над матеріалом вводу. Однак, коли певні ваги параметрів мережі прив'язані певним чином, механізм пам'яті в "End-to-End" мережах ідентичний механізму уваги, представленому тут, лише тим, що він здійснює декілька операцій над пам'яттю (оскільки він намагається інтегрувати інформацію з кількох речень).

Машини нейромережі Тьюрінга використовують подібну форму механізму пам'яті, але з більш складним типом адресації, яка використовує як на основі вмісту (як тут) так і адресну адресацію, що дозволяє мережі вивчати шаблон адресації для виконання простих комп'ютерних програм, таких як алгоритми сортування.

Цілком імовірно, що в майбутньому ми побачимо більш чітку відмінність між механізмами пам'яті та увагою, можливо, у відповідності до

зростаючого інтересу до навчання Neural Turing Machines, які намагаються вивчати схеми доступу до пам’яті для представлення зовнішніх інтерфейсів.

Ключовою ідеєю механізму уваги є встановлення прямих короткострокових зв'язків між ціллю та джерелом, звертаючи увагу на відповідний вихідний вміст під час перекладу. Хороший побічний продукт механізму уваги – це можливість легко візуалізувати матрицю вирівнювання між вихідними і цільовими пропозиціями.

У моделі vanilla seq2seq я передаю останній стан джерела від датчика до декодера на початку процес декодування. Це добре працює для коротких та середніх передбачень. Однак, для довгих передбачень, одна прихована фіксована величина стає інформаційно вузьким місцем. Замість того, щоб відкинути всі приховані стани, обчислені в джерелі RNN, механізм уваги забезпечує підхід, який дозволяє декодеру заглянути в них (розглядаючи їх як динамічну пам'ять вихідної інформації).

Таким чином, механізм уваги покращує переклад довгих речень. В даний час механізми уваги є стандартом defacto і успішно застосований до багатьох інших завдань (у тому числі створення заголовків зображення, розпізнавання мовлення та підведення підсумків тексту).

Мета завдання полягає в тому, щоб відповідати імовірнісній моделі, яка присвоює ймовірності реченням. Це відбувається, передбачаючи наступні слова в тексті, що дає історію попередніх слів. Для цього я використав набір даних Penn Tree Bank (PTB), який є популярним еталоном для вимірювання якості цих моделей, хоча він малий і відносно швидкий для тренувань.

Набір даних вже підготовлений і містить у загальному 10000 різних слів, включаючи маркер кінця речення, і спеціальний символ (<УНК>) для рідкісних слів. У reader.py я перетворюю кожне слово на унікальний цільний ідентифікатор, щоб нейронна мережа змогла легко обробляти дані.

ВИСНОВОК

Метою даного дослідження було прискорення процесу статистичного моделювання мов програмування, за рахунок використання фреймворку TensorFlow.

У роботі було проведено аналіз методів статистичного моделювання мов програмування та, як результат, наведено аргументи на користь використання нейронних мереж. Також проведено аналіз доступних програмних засобів для реалізації запропонованого методу та наведено аргументи на користь використання фреймворку TensorFlow. Після цього було проведено адаптацію оригінальних методів для використання з фреймворком TensorFlow. Розроблений метод дозволяє значно знизити тривалість процесів тренування класифікатора та текстової генерації вихідної програми, за рахунок проведення розподілених обчислень на TPU.

Для апробації запропонованого методу було розроблено програмний комплекс, який дозволяє проводити тренування мережі та генерувати вихідну програму за вхідним запитом користувача. Результати апробації доводять ефективність розробленого методу.

Апробація методу проводилася на підготовлених тестових даних, взятих з веб-сервісу навчання програмуванню Codewars. Практична цінність отриманих в роботі результатів полягає в тому, що запропонований метод та його програмна реалізація дають змогу більш швидше та продуктивніше проводити навчання нейронної мережі та отримувати більш якісну мовну модель з більш складною зв’язністю між окремими словами та реченнями у вхідному тексті, що дозволяє більш точно розпізнавати мови програмування та виокремлювати зміст написаного. В середньому процес навчання нейромережі на одному хмарному кластері TPUv2, який включає в себе 4

плати TPUv2 та 64GB RAM складає в 2,1 – 2,6 рази швидше, в порівняння з одним кластером GPU плат V100 від NVIDIA.

Але слід також зазначити, що при сьогоднішній ціновій політиці одна година використання хмарного кластеру с платами TPU є вищою в 2 рази порівнюючи з кластером плат GPU. Тому у фінансовому аспекті використовувати TPU плат також є вигіднішим у загальному випадку на 20%.

Подальшим напрямком досліджень може бути узагальнення та розширення запропонованого методу для автоматичного розпізнавання більш комплексних комп’ютерних програм з складнішою логікою.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Allamanis. M. Suggesting accurate method and class names / M. Allamanis, E. Barr, C. Bird, C. Sutton. / ESEC/FSE. 2015;

2. Cho. K. On the properties of neural machine translation: Encoder-decoder approaches / K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio / arXiv preprint; 1409.1259. 2014.

3. Chorowski. J. Attention-based models for speech recognition / J. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, and Y. Bengio. / arXiv preprint, 1506.07503; 2015.

4. Cozzie. A. Writing programs with natural language and examples / A. Cozzie, S. T. King. Macho. / Technical report, University of Illinois at Urbana-Champaign, 2012.

5. Flener. P. Inductive programming. Automated Softw. Engineering / P. Flener, D. / Partridge, 8(2):131–137, 2001.

6. Gulwani. S. Dimensions in program synthesis. / In Proc. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, 2010.

7. Helmuth. T. General program synthesis benchmark suite / T. Helmuth, L. Spector. / In Proc. Genetic and Evol. Comput. Conf. ACM, 2015.

8. Hochreiter. S. Long short-term memory / S. Hochreiter, J. Schmidhuber. / Neural Comput., 9(8):1735–1780, 1997.

9. Hyotyniemi. H. Turing machines are recurrent neural networks. / Proc. STeP, 1996.

10. Karpathy. A. Visualizing and understanding recurrent networks / A. Karpathy, J. Johnson, F. Li. / arXiv preprint, 1506.02078, 2015.

11. Knoll. R. Pegasus: First steps toward a naturalistic programming language / R. Knoll, M. Mezini. / In OOPSLA, 2006.

12. Kumar. A. Ask me anything: Dynamic memory networks for natural language processing / A. Kumar, O. Irsoy, J. Su, et al. / arXiv preprint, 1506.07285, 2015.

13. LeCun. Y. Deep learning / Y. LeCun, Y. Bengio, G. Hinton. / Nature, 521(7553):436–444, 2015.

14. Manna. Z. A deductive approach to program synthesis / Z. Manna and R. Waldinger. / ACM Trans. Programming Languages and Syst., 2(1):90–121, 1980.

15. Mou. L. TBCNN: A tree-based convolutional neural network for programming language processing / L. Mou, G. Li, Z. Jin, L. Zhang, T. Wang. / AAAI Workshop, 2015.

16. Mou. L. Building program vector representations for deep learning / L. Mou, G. Li, Y. Liu, H. Peng, Z. Jin, Y. Xu, L. Zhang. / arXiv preprint, 1409.3358, 2014.

17. Pascanu. R. On the difficulty of training recurrent neural networks / R. Pascanu, T. Mikolov, Y. Bengio. / arXiv preprint, 1211.5063, 2012.

18. Sutskever. I. Sequence to sequence learning with neural networks / I. Sutskever, O. Vinyals, Q. Le. / In NIPS, 2014.

19. Weimer. W. Automatically finding patches using genetic programming / W. Weimer, T. Nguyen, C. Le Goues, S. Forrest / In ICSE, 2009.

20. Zaremba. W. Learning to execute / W. Zaremba, I. Sutskever / arXiv preprint, 1410.4615, 2014.